

DISK  
CLOSED

# AC's TECH AMIGA

For The Commodore

Volume 3 Number 3  
US \$14.95 Canada \$19.95

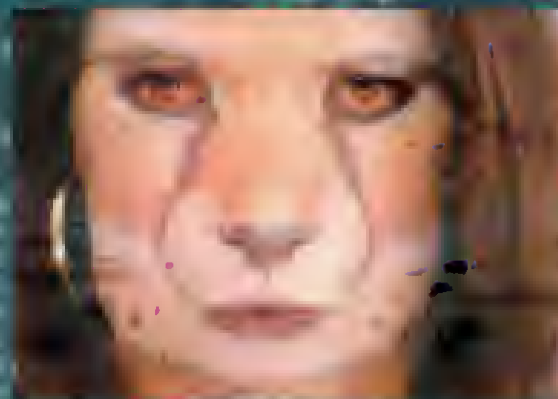
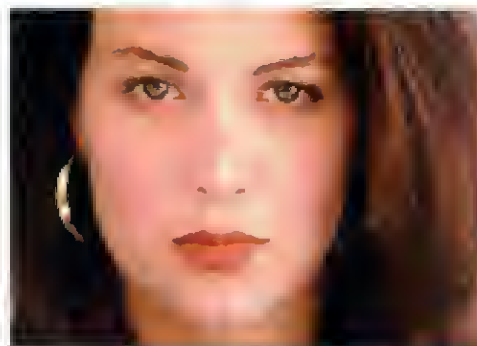
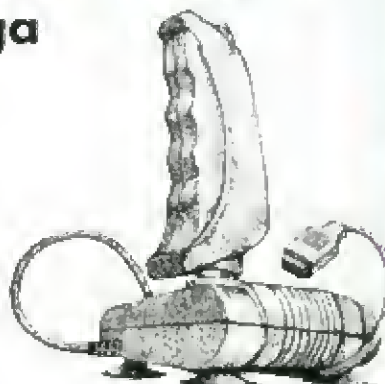
## Morphing

- REXX Rainbow Library
- Programming the Amiga in Assembly Language: Making Waves
- All You Ever Wanted to Know About Morphing
- Create Your Own Custom 3-D Graphics Package, Part I
- StateMachine
- Programming in PostScript

### • Hardware Project:

Build a Second Joystick Port For Your Amiga

- Plus  
much more!





## FAST AND POWERFUL PRODUCTS FOR AREXX

Compile your ARExx programs with the REXX PLUS COMPILER and they will execute up to 18 times faster. The Intuition Interface allows even the most novice user to execute their programs at warp speed. Explicit error messages make debugging a breeze. The REXX PLUS COMPILER generates a listing that is easier to read than the original source. The listing contains nesting levels, flagged comments, a symbol table and a complete cross reference. Version 1.3 is a major upgrade that generates 40 to 60% smaller programs. All REXX RAINBOW LIBRARY SERIES functions can be included as part of the language.



**Don't just take our word for it, here is what some of the experts have to say about the REXX PLUS COMPILER...**

"...A SIGNIFICANT NEW PRODUCT WHICH ALL AREXX PROGRAMMERS SHOULD HAVE."

*Amazing Computing, June 1992*

"...THE AUTHORS HAVE IT RIGHT... IT COULD WELL BE A FUTURE AMIGA CLASSIC."

*Amiga Computing UK, November 1992*

"...IS A WELL-DESIGNED UTILITY THAT DOES ITS UTMOST TO SUPPORT THE COMPLETE AREXX ENVIRONMENT IN A TRANSPARENT FASHION."

*Amiga World, September 1992*

"...DOES THE JOB AND DOES IT WELL, EVEN ELEGANTLY."

*Jump Disk, June 1992*

**NEW**

## REXX *Rainbow Library* S E R I E S

The REXX RAINBOW LIBRARY SERIES is a complete product line of support libraries designed specifically for use with ARExx. Each volume in the Series contains functions dedicated to a specific subject. The first volume in the series is the Stem/Array functions. It provides over 100 functions to manipulate single dimension arrays, which simplify ARExx arrays, Compound Symbols, Pointers and Subscripts. The functions include string manipulation, mathematical and scientific calculations and file access. Also included is the AssignArray() function which assigns/retrieves arrays from/to other ARExx programs. With this function you can build your own single or multiple dimension array functions. Tutorials and examples are used throughout the manual. The REXX RAINBOW LIBRARY SERIES requires ARExx and works with or without REXX PLUS.

**DEMO DISK AVAILABLE**



**Dineen Edwards Group**

19785 W. 12 Mile Rd., Suite 305  
Southfield, MI 48076-2553

**313-352-4288**

Amiga Ben is a registered trademark of Commodore Business Machine. ARExx is a registered trademark of Wishful Thinking.

# Contents

Volume 3, Number 3

AC's TECH/AMIGA

---

- 5 **Rexx Rainbow Library** *by Merrill Callaway*  
*A review of this handy ARexx library from Dineen Edwards.*
- 10 **Programming the Amiga in Assembly Language:  
Making Waves** *by William P. Nee*  
*Learn new and interesting techniques in this latest installment in the  
Assembly language series.*
- 19 **All You Ever Wanted to Know About Morphing  
But Were Afraid to Ask** *by Bruno Costa & Lucia Dorsa*  
*Two leading graphics programmers clue you in to the morphing  
scene.*
- 27 **Have Your Own Custom 3-D Graphics Package, Part I**  
*by Laura Morrison*  
*Don't like the packages available? Create your own.*
- 43 **StateMachine** *by J.T. Steichen*  
*Create a finite state machine with a Graphical User Interface rather  
than generating the program by hand.*
- 54 **Programming in PostScript** *by Dan Weiss*  
*A look inside the workings of the PostScript language.*
- 66 **Need A Second Joystick Port?** *by Jaques Hallée*  
*With this creative hardware project, you can add a joystick  
switchbox to your Amiga.*

## Departments

---

- 3 **Correspondence**
- 48 **List of Advertisers**
- 49 **Source and Executables ON DISK!**
- 77 **AC's TECH Back Issues!**

```
printf("Hello");
```

```
print "Hello"
```

```
JSR printMsg
```

```
say "Hello"
```

```
writeln("Hello")
```

---

**Whatever language you speak, AC's TECH provides a platform for both gaining insight and sharing information on its most innovative implementation for the Amiga. Why not see if your latest programming endeavor can help a fellow Amiga user expand upon his or her vocabulary? To be considered for publication in AC's TECH, submit your technically oriented article (both hard copy & disk) to:**

AC's TECH Submissions  
PiM Publications, Inc.  
P.O. Box 2140  
Fall River, MA 02722-2140

## AC's TECH / AMIGA

### ADMINISTRATION

Publisher:	Joyce Hicks
Assistant Publisher:	Robert J. Hicks
Administrative Asst.:	Donna Viveiros
Circulation Manager:	Doris Gamble
Asst. Circulation:	Traci Desmarais
Traffic Manager:	Robert Gamble
Marketing Manager:	Ernest P. Viveiros Sr.

### EDITORIAL

Managing Editor:	Don Hicks
Editor:	Jeffrey Gamble
Hardware Editor:	Ernest P. Viveiros Sr.
Senior Copy Editor:	Paul Larrivee
Copy Editor:	Elizabeth Harris
Video Consultant:	Frank McMahon
Illustrator:	Brian Fox

### ADVERTISING SALES

Advertising Manager: Wayne Arruda

1-508-678-4200  
1-800-345-3360  
FAX 1-508-675-6002

AC's TECH For the Commodore Amiga™ (ISSN 1053-7929) is published quarterly by PiM Publications, Inc., One Cunant Road, P.O. Box 2140, Fall River, MA 02722-2140.

Subscriptions in the U.S., 4 issues for \$44.95; in Canada & Mexico surface, \$52.95; foreign surface for \$56.95.

Application to mail at Second-Class postage rates pending at Fall River, MA 02722.

POSTMASTER: Send address changes to PiM Publications, Inc., P.O. Box 2140, Fall River, MA 02722-2140. Printed in the U.S.A. Copyright © 1993 by PiM Publications, Inc. All rights reserved.

First Class or Air Mail rates available upon request. PiM Publications, Inc. maintains the right to refuse any advertising.

PiM Publications, Inc. is not obligated to return unsolicited materials. All requested returns must be received with a Self-Addressed Stamped Mailer.

Send article submissions in both manuscript and disk format with your name, address, telephone, and Social Security Number on each to the Editor. Requests for Author's Guides should be directed to the address listed above.

AMIGA™ is a registered trademark of Commodore-Amiga, Inc.



# Correspondence

*The following letter is from Paul Castonguay. It answers some questions raised by readers regarding AmigaBASIC subprograms.*

## AmigaBASIC SUBPROGRAMS

In all fairness to the folks at Microsoft, I would like to point out an inaccurate statement you (Paul Castonguay) keep making in your articles. In them you state that an AmigaBASIC SUBPROGRAM cannot invoke another SUBPROGRAM. I don't know where you got this idea, but it is not true. It does seem to be a widely held myth, however, I've seen this statement in print before. AmigaBASIC does have its shortcomings, of course, but to be fair about it, this isn't one of them.

## Mr. Probst, Columbia, MD

You're right, I think. Yes, an AmigaBASIC SUBPROGRAM invoking another SUBPROGRAM does seem to work, and yes, it is a widely held myth. I did some research and uncovered the following on page 442 of: "Advanced AmigaBASIC", by Tom R. Hallhill & Charles Brannon, COMPUTE! Publications, 1986, which may have led me astray:

"There are a few disadvantages of subprograms, ... They can't be nested, and one subprogram can't call another subprogram."

Since I was an early user of the Amiga, I suspect that I had the misfortune of reading that before I thought of trying it on my own. That's too bad. I have published many a program that would have been better written if only I had known that I could do this. Indeed, I even curtailed the publishing of articles about general purpose libraries of SUBPROGRAMS simply because I thought that they could not be invoked reliably from anywhere in a program.

I tested your correction and found it to work, as you have reported. The only operational restriction I could find was that no SUBPROGRAM could be invoked twice, either by itself recursively, or by another through some recursive connection of SUBPROGRAMS.

At the same time, I checked my AmigaBASIC manuals but nowhere is it explicitly stated that a SUBPROGRAM really can invoke another, although the implication is strong that it can. Is that enough to openly say that it can be done reliably? Has this been thoroughly tested? It's too bad that Microsoft does not continue to support the language, otherwise we could all find out for sure. It's possible that there are limitations to this. For example, many early versions of

BASIC allowed only a limited number of levels of SUBROUTINE or SUBPROGRAM calls, some as little as eight.

Since you have pointed this out I will no longer make the claim that an AmigaBASIC SUBPROGRAM cannot invoke another SUBPROGRAM. However, neither will I say that you can, at least not until I find out for sure how reliable it is to do so.

Thank you for the correction. Alas, it may be too late anyway, since AmigaBASIC is no longer bundled with the Amiga. It is unfortunate that over the years no Amiga users have chosen to write an article to clear up this popular misconception about AmigaBASIC.

—Paul Castonguay

---

*The following is a correction to the listing in Bill Nee's Assembly Language & Computer Simulations article, AC TECH 3.2.*

```
;LISTING1
equates:
depth      = $4
jam2       = $1
mousebuttons = $8
borderless = $800
ww.screen  = $2e
ww.rport   = $32
nw.screen  = $1c
customscreen = $f
activate   = $1000
rmb        = $10000
public     = $1
chip       = $2
fast       = $4
clear      = $10000

offsets:
;exec
openlibrary = -552
closelibrary = -414
allocmem    = -198
freemem     = -210
forbid      = -132    ;no registers used
permit      = -138    ;no registers used
;intuition
```

```

openScreen      96
closeScreen     66
openWindow      204
closeWindow     72
viewportAddress -30 ;a0=window
;graphics
setdramd        354
loadrgb1        -182 ;a0=vp.ad;col=able,d0=#pens
setapen         -342
writepixel      -534

```

```

sum equ d0
across equ d6
down equ d7
bb equ d4
aa equ d3
len 32*4 ;array 11*5 for AMIGA 3000
lenm1 = len-1
mlenm1 = -1-lenm1
lenm2 = len-2
xoff = (320-len)/2 ;to center display
yoff = (300-len)/2 ;to center display
lenp1 = len-1
lenm3 = len-3
nlen = 1*len
byte = (yoff*320+xoff)*8 ;byte containing yoff/xoff
wpl = len/4-1 ;long words per line
byteoff = 3*xoff+8 ;offset to next byte
size = len*len
;k1 equ 1 ;default
;k2 equ 3 ;default
;rg equ 4 ;default

```

#### Macros:

```

syslib macro ;routine
    movea.l 4,a6
    ;a1 = 0000
    endm

openlib macro ;iname, location, BEQ, 11
    lea 11,a1
    moveq #0,d0
    syslib openlib,ary
    move.l d0,a2
    beq a3
    endm

arlib macro ;routine
    movea.l intbase,ar
    ;ar = 0000
    endm

openScreen macro ;parameters, screen, BEQ=0
    lea a1,d0
    intlib openScreen
    move.l d0,a2
    beq a3
    endm

openWindow macro ;parameters, window, BEQ=0
    lea a1,d0
    move.l screen,nw,screen+0
    intlib openWindow
    move.l d0,a2
    beq a3
    endm

```

```

gfxlib macro ;routine
    movea.l gfxbase,ah
    ;ar = 0000
    endm

test macro ;top, left, right, bottom
    beq.s healthy,9
    cmpi.b #50,d0
    bcc.s 11,9
    moveq #1,d0
    bra 1000,done,9

healthy,9
    moveq.b #1,a3
    moveq.b #0,b5
    h1,9
    move.b 11040,d1
    beq.s h2,9
    cmpi.b #50,d1
    beq.s h1a,9
    addq.b #1,aa
    bra.s h1a,9
    h1a,9
    addq.b #1,bb
    h2,9
    move.b 12040,d1
    beq.s h3,9
    cmpi.b #50,d1
    beq.s h3a,9
    addq.b #1,aa
    bra.s h4,9
    h3a,9
    addq.b #1,bb
    h4,9
    move.b 13040,d1
    beq.s h4a,9
    cmpi.b #50,d1
    beq.s h4a,9
    addq.b #1,aa
    bra.s h5,9
    h4a,9
    addq.b #1,bb
    h5,9
    moveq #0,d1
    move.b k1,d1
    divu d1,d0
    moveq #0,d1
    move.b k2,d1
    divu d1,bb
    addi.b bb,aa
    move.b aa,d0
    bra.s 1000,done,9
    h1,9

```

(continued to page 58)

# REXX Rainbow Library

## Volume of Stem/Array Functions

*by Merrill Callaway*

**T**he REXX Rainbow Library Series marks a new and useful effort by the Dineen Edwards Group, who brought us the REXXPlus Compiler. This new series of ARexx-specific libraries is arguably more important than the REXXPlus Compiler in the scheme of things, but the REXXPlus Compiler adds an interesting connection to the Rainbow Libraries. As you may know, if you program an application that uses functions from a shared library, and attempt to export or distribute it to an Amiga that lacks that library, then it will not run. The other computer must have the library aboard. In the case of freely distributable libraries, such as Willy Langeveld's `rexarplib.library`, you simply send along a copy of the library. But Dineen Edwards is commercial, and you cannot distribute any of the Rainbow Libraries without a license. However, because of the compiler options in REXXPlus, you may "build in" the functions from a Rainbow library at compile time and the compiled product will be distributable without further license, and the library need not be present on the other system. Dineen Edwards has changed their policy and absorbed the former \$10 licensing fee for distribution of REXXPlus compiled programs into the price of the REXXPlus Compiler. A complete set of routines and libraries are included in the Rainbow Library Disk for REXXPlus (RP) support. I did not compile the examples given here because of a problem in my compiler that was not fixed in time. By the time you read this, there will be a newer version of REXXPlus. I found a glitch in a pre-release Stem/Array Function, and Mark Edwards of Dineen Edwards called in my fix data (a CLzap program comes on disk) within eight hours. That's service! The Stem/Array Functions worked very well.

## Adding Stem/Array Library to ARexx Library List

The first of the Rainbow Library Series volumes contains 107 array functions, much like those in the standard ARexx function set, except that these functions act on one or two single-dimensional arrays, and usually output to a third "answer" array. To add the Stem/Array (SA) library to your ARexx library list, from a Shell prompt>

```
CALL ADDLIB("rexarray.library",0,30,0)
```

where the library name is case sensitive and the keywords are "priority," "offset," and "version" respectively. Inside an ARexx program the following code accomplishes the same thing:

```
CALL ADDLIB("rexarray.library",0,30,0)
```

## Syntax

The SA functions share a similar syntax. If there is a similar ARexx function, the SA name is a composite of the regular ARexx function name with "array" concatenated to it. For example the ARexx function LASTPOS() becomes LASTPOSarray(). SA functions that don't share a corresponding regular ARexx function have an intuitive name such as ADDarray() which ADDs two single dimensional arrays

```
/* pgm fragment */
count=5
start=1
m=ADDarray(ANS,numA,numB,5,1)
say m
/*
** say m returns 5 for the number of compound symbol tokens
assigned
** These lines also have correct syntax:
** m=ADDarray(ANS,numA,numB,count,start)
** m=ADDarray(ANS,numA,numB,5,1)
** CALL ADDarray(ANS,numA,numB,...)
** m=ADDarray(ANS,numA,numB)
** The above may not return the same values as they depend on what
the
** VALUES OF numA,0 and numB,0 are, although they are OK in syntax.
*/
```

## Syntax Conventions

The answer array, if we were to put in a DO loop to write it to the console would have elements 1 through 5 added from the two arrays. If one of the elements is uninitialized, then it is set equal to 0 in a numeric array function or the null string "" in a string array function. This is normally OK, but you will get an error if you use the DIVIDEarray() function and try to divide by either 0 or a compound symbol token that is not initialized. Another convention is that if the COUNT is left off, then the COUNT=MAX(numA,0,numB,0) in the case of two input arrays present, or just the 0th element of numA, if

**A complete set of routines and libraries are included in the Rainbow Library Disk for RexxPlus support.**

together. The arguments, separated by commas, to the SA functions have a family resemblance, too. The "answer" array always comes first, the actual argument being the STEM or name for the answer array. If you leave off the period (.) at the end of the STEM, SA will add one for you. In general all arrays are named by their stems, the name of the array with exactly one period at the end. For purposes of explanation, we will call the answer array's stem ANS, but of course you may name it anything. As soon as nodes index an array, the stem.nodeA.nodeB.nodeC etc. becomes a compound symbol token in ARexx-speak. There are up to two INPUT arrays which here we will call numA. and numB. (even though they can be string arrays as well—of course they may be any stem name at all in your program). The general pattern of argument syntax is ANS.numA.numB., but there are some exceptions. In general, the last two arguments are the COUNT and the START numbers. COUNT and START are optional and there are conventions followed if we omit them, as we'll see later. Assume we have two arrays numA. and numB. and we want to add them element by element, starting at element 2 and adding 5 elements in a row and assign the results of the addition into an array called ANS., then the ADDarray() function in a program would look like this:

only one is present. If either numA,0 or numB,0 (if numB. is present as an argument) is uninitialized then COUNT=0 is assigned. If numA,0 or numB,0 is assigned to a non-numeric value, then a numeric conversion error results. If the start argument is omitted, then START=1 becomes the convention. I think there ought to be a convention to take care of division by zero in the case of an uninitialized compound symbol token in numB., as the manual states that making an uninitialized compound symbol 0 was to avoid numeric errors! Only if you actually go and put in a 0 into numB should there be a numeric error.

## A Syntax Exception

One major syntax exception is ASSIGNarray() which is the most powerful function of the lot. Plain ARexx does not allow you to pass intact arrays to an exterior ARexx program from a main program, nor does it allow you to get an array back to the calling program from the subprogram. The only way to pass an intact array in ARexx is to an INTERIOR function, called a PROCEDURE. Here, you may pass the STEM and the COUNT as arguments, and the entire array will go over. With ASSIGNarray(), you may not only obtain intact arrays from the calling program inside an EXTERIOR function program, you can obtain whole arrays from the program that called the program that called the one you are in! ASSIGNarray() lets you specify how many levels if any you wish to go back to obtain its values! This argument is called NEST LEVEL (NL). Instead of calling it the input stem or array,



the SA manual calls it the VALUE stem. The output array is called the ASSIGN stem by SA. I prefer to think: FROM the value array TO the assign array. In FROM and TO terms, the argument syntax is

```
meASSIGNarray(TO_NL,TO Stem,,FROM Stem,,FROM_NL,Count,Start)
SAY m
```

Any function may be CALLED to discard the return value. Here the function returns the number of compound symbols it assigned in the simple symbol, m. Count and Start may be omitted and the syntax will follow the above cited conventions. The Nest\_Level (NL) follows this convention: 0 means THIS program; 1 means ONE program back (the one that called this one); 2 means the program which called the one which called this one; etc. Entire arrays may be assigned by ASSIGNArray() within the same program (both TO\_NL=0 and FROM\_NL=0), or more commonly, you will code two calls to ASSIGNArray(), one to GET the array from a calling program and another to PUT the processed array back to the calling program. The example program, SANodupes.rexx calls a sort routine, SASort.rexx which uses ASSIGNArray() to get the array LIST from the calling program, sort this array, and then put the sorted array back to SANodupes.rexx by using another ASSIGNArray() with the four TO and FROM arguments, reversed. The original STEM, COUNT, and START are passed to the external function, SASort.rexx in the normal way as arguments.

The second example shows SATester.rexx, a program to try out the functions and study the way the arrays are processed. It uses an INTERPRET instruction as a basis for testing the functions which are built up via the powerful string handling ARExx exhibits. Ironically, the INTERPRET instruction cannot be compiled with RexxPlus successfully, as it is too self referential for cost effective system requirements. SATester.rexx will work very well with the resident process, rexxmast, however. You will need the Rainbow Stem/Array Library to run these.

I look forward to the other Rainbow Series Libraries, as they will undoubtedly enrich my ARExx capabilities. If you plan to distribute programs using the Rainbow Series, I'd suggest buying the RexxPlus Compiler, as that is the most economical way to export these functions without further licensing.

## SAnodupes.rexx

```
/*
** SANodupes.rexx
** SASort.rexx uses Rainbow Series Stem/Array
** This pgm calls the sort routine SASort.rexx as an
** exterior
** function, where the array here is assigned there via
** AssignArray().
** The returned value is the number of words sorted. The
** external
** function program alters the values of the array from
** this
** program, so that the final array written at the end of
```

this program

```
** is the sorted array EVEN THOUGH it is the "SAME" ARRAY!
** copyright 1992, 1993 by Merrill Callaway
**
```

```
SAY 'Input filename and path.'
FULL infile
CALL TIME('R')
rcode=20
IF OPEN('textfile',infile,'READ') THEN DO
  rcode=0

  /* initialize an entire array with one line! */
  listed.=0

  m=1
  i=1
  DO WHILE -EOF('textfile')
    line.i=READLN('textfile')

    IF line.i == '' THEN DO
      j=1
      DO WHILE line.i == ''
        PARSE VAR line.i word.i.j line.i
        word=word.i.j
        /* NOTE! You cannot use an array as a NODE! */
        /* Get rid of punctuation at end and beginning
        of words. */
        DO FOREVER
          IF DATATYPE(word,'N') THEN LEAVE
          IF -DATATYPE(RIGHT(word,1),MIXED) THEN DO
            IF LENGTH(word)>1 THEN
              word=LEFT(word,LENGTH(word)-1)
            ELSE LEAVE
          END
          IF -DATATYPE(LEFT(word,1),MIXED) THEN DO
            IF LENGTH(word)>1 THEN
              word=RIGHT(word,LENGTH(word)-1)
            ELSE LEAVE
          END
        END
        IF
          DATATYPE(LEFT(word,1),MIXED)&DATATYPE(RIGHT(word,1),MIXED),
          THEN LEAVE
        IF LENGTH(word)=0 THEN LEAVE
        END
        /* end punctuation removal */

        IF word='' THEN ITERATE

        /* testing of "random access" array */
        IF listed.word THEN ITERATE
        listed.word=1 /* boolean "yes" */
        list.m=word
        m=m+1
        j=j+1
      END
      i=i+1
    END
    i=i+1
  IF line.i == '' THEN i=i-1
  END
ELSE SAY 'Could not open your file!'

  /* Adjust the count. */
  m=m-1
```

```

* Call the exterior function program. */
k=SASort(list,,m,1)
SAY k 'entries sorted.'

* Write the sorted array to the console. */
DO j=1 TO k
  SAY list.j
END

* For time comparisons of this pgm with others. */
SAY 'Elapsed time is 'TIME('E')' seconds.'

EXIT rcode

```

## SASort.rexx

```

*
** SASort.rexx
** An external function which may be called from any
** program with a list in array form. The stem, the
** count,
** and the start are passed as arguments, picked up by
** this
** program, and by means of AssignArray() are reconsti-
** tuted
** into an array to be sorted. At finish, the array is
** assigned
** back to the one from which it came.
*****
** YOU MUST HAVE Rainbow Series Stem/Array
** rexxarray.library
** TO USE THIS PROGRAM! It is available from: Dineen
** Edwards Group.
** (313) 352-4288 OR this pgm must have been compiled by
** RexxPlus.
** This program copyright 1992, 1993 by Merrill Callaway
*

* Load the rexxarray.library */

IF SHOW('L','rexxarray.library') THEN DO,
  CALL ADDLIB('rexxarray.library',0,-30.0)
  IF SHOW('L','rexxarray.library') THEN DO
    SAY 'rexxarray.library failed to open.'
    EXIT 5
  END
END

* The Shell Sort. Feed the array to Shell Sort directly!
*/

* get the name of the stem, the count m, and the start
value */
PARSE ARG stem,m,start

* guarantee list. is uninitialized if compiled with
RexxPlus *
DROP list.

```

```

/*
** Call Rainbow Stem/Array library AssignArray() function
** Assigns stem from the calling program at level 1 nest
to list. stem
** in this program level 0 for m count starting at start.
** Syntax for AssignArray() Assign TO and Assign FROM:
** (TO_LEVEL,TO_STEM,FROM_STEM,FROM_LEVEL,COUNT,START)
** 0 means THIS PGM; 1 means ONE LEVEL UP of calling pgms.
*/

```

```
CALL AssignArray(0,list,,stem,1,m,start)
```

```
listlength = m
```

```

span = 1
DO WHILE (span < listlength); span = span * 2; END
DO WHILE (span > 1)
  span = span % 2
  numpairs = listlength - span
  DO node = 1 TO numpairs
    nextnode = node + span
    IF list.node > list.nextnode THEN
      DO
        store = list.nextnode
        list.nextnode = list.node
        DO bubpos = node-span TO 1 BY -span WHILE (store
< list.bubpos)
          nextnode = bubpos + span
          list.nextnode = list.bubpos
          END bubpos
        bubpos = bubpos + span
        list.bubpos = store
        END
      END node
  END

```

```
/* the end of the shell sort of the words */
```

```
/* Assign TO stem there (1), FROM list. here (0), for
listlength.start */
```

```
CALL AssignArray(1,stem,list,,0,listlength.start)
```

```
EXIT listlength /* entire exterior function returns how
many words sorted */
```

## SAtester.rexx

```

/* SAtester.rexx to test the S/A functions */
SIGNAL ON BREAK_C
START:
SAY 'CTRL-C to quit.'
SAY 'Enter a line of separate numbers as Array A'
SAY '[Rtn] for a default Array A.'
PULL lineA

IF lineA = '' THEN lineA='2 5 8 10 0.5'
SAY 'Array A is'
SAY lineA

```

```

SAY 'Enter a line of separate numbers as Array B'
SAY '[Rtn] for a default Array B.'
PULL lineB

IF lineB = '' THEN lineB='1 5 4 5 0.1'
SAY 'Array B is'
SAY lineB

i=0
DO WHILE lineA ~= ''
  PARSE VAR lineA numA.i lineA
  i=i+1
END

j=0
DO WHILE lineB ~= ''
  PARSE VAR lineB numB.j lineB
  j=j+1
END

w=MAX(i-1,j-1)

SAY 'Enter Stem/Array function name you wish to test. NO
it.'
PULL saf
funname=saf
saf=saf'('
SAY 'Enter number of arguments.'
PULL numarg

SAY 'Use numA & numB as INPUT array names.'
SAY
DO a=1 TO numarg
  ending=''
  rtn='[Rtn]='
  SELECT
    WHEN a=1 THEN ending='ANS'
    WHEN a=2 THEN ending='numA'
    WHEN a=3 THEN ending='numB'
    OTHERWISE rtn=''
  END

  SAY 'Enter argument No.'a rtn||ending
  PARSE UPPER PULL argument.a '.'
  IF argument.a='' THEN argument.a=ending
  IF a<numarg THEN saf=saf||argument.a','
    ELSE saf=saf||argument.a')'
  SAY saf
END

SAY
SAY 'OK to interpret? N=exit.'
PULL ok

IF ok='N' THEN EXIT 5
saf='m='saf

INTERPRET saf

SAY m' array elements processed.'

nxt1=numarg-1

```

```

strt=argument.numarg
count=argument.nxt1
IF ~DATATYPE(count,'W') THEN count=MAX(numA.0,numB.0)
IF ~DATATYPE(strt,'W') THEN strt=1

```

```

SAY 'Count='count
SAY 'Start='strt

```

```

IF funname='ASSIGNARRAY' THEN DO
  ans=argument.1||'.'
  SAY 'Answer_Stem is 'argument.1
  END
ELSE DO
  ans=argument.2||'.'
  SAY 'Assign_Stem is 'argument.2
  END

```

```

SAY saf
SAY funname 'answer table for' ans
DO k=0 TO w
  SAY ans||k'='VALUE(ans||k) 'numA.'k' 'numA.k
  'numB.'k'='numB.k
  END

```

```

DROP numA.
DROP numB.
DROP ans

```

```

$SIGNAL START

```

```

BREAK_C;
EXIT 0

```



Please write to  
 Merrill Callaway  
 c/o AC's TECH  
 P.O. Box 2140  
 Fall River, MA 02722

# Programming the Amiga

## ANOTHER SIMULATION

In this article I'll show you another computer simulation using the Amiga. You see this simulation in action whenever you drop a stone in the water, watch a heart-beat, or stir your coffee. The effect produced is wave action—waves that move out from the disturbance and interact with whatever they touch. These waves could be electrical impulses along heart muscles, rust spreading on metal, or simply ripples in a pond. Using a principle developed by John J. Tyson of V.P.I., we'll simulate wave motion in an excitable media; water is not an excitable media, but nerve cell membranes and oxidation are. Excitable media keeps sending out waves over relatively long distances.

Whatever media we use, it will be described as being in an excitation state ( $U$ ) or a recovery state ( $V$ ). The  $U$  state will only have two values - 0 (unexcited cell) or 1 (excited cell). The recovery state  $V$  will range from 0 to  $V_{MAX}$  which throughout these examples will always be 100. In simple terms, consider one cell in the media; at any given time it has a  $U$  and  $V$  value. To determine its values for the next generation look first at its  $U$  value. When  $U=1$ ,  $V$  increases by a fixed amount called  $GUP$ ; when  $V$  reaches  $V_{MAX}$ ,  $U$  becomes 0. If  $U$  is initially 0,  $V$  will decrease by an amount called  $GDOWN$  until it reaches 0. When both  $U$  and  $V$  are 0, the media is stable.

If these were the only variables, a media would quickly settle down to its stable state and wave motion would cease. We can add other variables however, that will simulate specific reactions and produce more realistic demonstrations. First we'll let a cell be partially affected by cells around it; these cells are called the neighborhood. If we go out a distance of  $R$  (generally  $R=3$ ) in all directions from a given cell then our neighborhood consists of 49 cells in a square centered over a given cell; if  $R=2$  the neighborhood is 25 cells. The size of any neighborhood is  $(2R+1)(2R+1)$ .

The variable  $V_{EXC}$  will represent a sufficiently recovered state. An unexcited cell ( $U=0$ ) can become excited ( $U=1$ ) at the next generation if  $V$  is between 0 and  $V_{EXC}$  and the number of excited cells in a neighborhood ( $E$ ) exceeds a variable  $K_{EXC}$ . The variable  $V_{REC}$  represents a

b y W i l l i a m P. N e e

sufficiently unexcited  $V$  state. An excited cell ( $U=1$ ) can become unexcited ( $U=0$ ) at the next generation if  $V$  is between  $VREC$  and  $VMAX$  and the number of unexcited cells in the neighborhood ( $EE$ ) exceeds a variable  $KREC$ . For all other cases we'll only add  $GUP$  or subtract  $GDOWN$  from  $V$  depending on the  $U$  value. Also, if  $V=VMAX$  then an excited cell ( $U=1$ ) will automatically become unexcited ( $U=0$ ).

#### GUP, GDOWN

$GUP$  and  $GDOWN$  depend on the local kinetics of the excitable media. These values may represent the oxidation of iron, diffusion in a membrane, etc. While their values could be obtained from a chemistry book, we'll be more interested in programming the simulation and experimenting with different values. To summarize the rules at this point:

```
IF D=0
  IF V<VREC THEN NEWV=V-GDOWN:IF NEWV<0 THEN NEWV=0:NEWU=0:LOOP
  COMPUTE THE NEIGHBORHOOD OF EXCITED CELLS (E)
  * IF E<KEXC THEN NEWU=1:NEWV=V:LOOP
  IF E<KEXC THEN NEWV=V-GDOWN:IF NEWV<0 THEN NEWV=0:NEWU=0:LOOP
IF D=1
  IF V<VREC THEN NEWV=V-GUP:IF NEWV<VMAX THEN NEWV=VMAX:NEWU=1:LOOP
  * IF V<VMAX THEN NEWU=0:NEWV=VMAX:LOOP
  COMPUTE THE NUMBER OF UNEXCITED CELLS (EE)
  * IF EE<KREC THEN NEWU=0:NEWV=V:LOOP
  IF EE<KREC THEN NEWV=V-GUP:IF NEWV<VMAX THEN NEWV=VMAX:NEWU=1
*CHANGE IN D STATUS
```

# in Assembly Language

Notice that there is only one way an unexcited cell state can go from 0 to 1, but there are two ways an excited cell state may become unexcited. This process gradually dampens the initial disturbance smoothing things out just as in real life.

Initial values of U and V are stored in two separate arrays; new U and V values will be stored in two more arrays. After all the new values for each cell have been computed, they are transferred back to the original array. The display is colored using the U values of 0 or 1 for each cell and point on the screen.

If you wrote the program at this point there would be two problems. First, the waves don't appear to be waves; they're more like large blobs and most of them don't interact. It's not too realistic and definitely not interesting to watch. The second problem is speed—or rather the lack of it. Computing that neighborhood of 49 squares takes too long and slows everything down especially on the Amiga 500. And if we use a larger neighborhood, say R=6, that's 169 cells to evaluate!

## BETTER K VALUES

Let's correct the first problem by making a refinement to the K values. As an unexcited cell's V values get closer to 0 it takes fewer and fewer neighbors to change its U state to 1. Conversely, as an excited cell's V values get closer to VMAX it takes more neighbors to change

The sum of the neighborhood around a cell can be computed using only four values from SQUARE. E at location (X,Y) is equal to

$$\text{SQUARES}(X+X, Y+Y) + \text{SQUARES}(X-R-1, Y-R-1) + \text{SQUARES}(X-R-1, Y+R+1) + \text{SQUARES}(X+R, Y-R-1).$$

Now the number of steps required to compute E is not dependant on R. The extra time required to fill the initial SQUARES array is much less than the time saved in computing E with four steps versus computing E with 49 steps.

The demo for the program fills the left-half of a square with excitable media (U=1); the right-half is unexcited (U=0) and all V values are 0. Waves will begin to come out from the excited portion of the media and as time progresses they will begin to interact with themselves, collide, and form new waves or spirals. Use the initial values given and then try some of the ones at the end of the program.

## THE LISTING

Let's look at the machine language program (Listing1) in detail. After the usual equates and offsets the variables SUM, TESTV, and E are equated to specific data registers. Notice that SUM and TESTV are both equated to register d7; as long as these variables are not used

its V state to 0. We'll reflect this by spreading the K values out in arrays as V ranges from 0 to VMAX based on the formulas:

$$\begin{aligned} \text{KEXC}(V) &= \text{KREC} + (R/2R+1) - \text{KREC} * V/\text{VREC} \\ \text{KREC}(V) &= \text{KREC} + (R/2R+1) - \text{KREC} * (VMAX-V)/VMAX \end{aligned}$$

Now the E value will be compared to KEXC(V) and EE will be compared to KREC(V). Remember that VMAX=100. Let's look at some actual values. One example in the listing is:

$$R=3, \text{VREC}=70, \text{VMAX}=65, \text{GVF}=20, \text{UDOWN}=5, \text{KREC}=5, \text{KEXC}=1, \text{VMAX}=100$$

Using the above formulas as V ranges from 0 to VMAX, KEXC(V) would range from 1 to 31; KREC(V) ranges from 53 to 5.

## A QUICKER ARRAY

We could still check all 49 squares in the neighborhood to compute the E or EE value, but there is a quicker way. This new way, however, requires a fifth array called SQUARES. Any point in this array, say SQUARES(I,J), equals the corresponding U value U(I,J)+SQUARES(I-1,J)+SQUARES(I,J-1)-SQUARES(I-1,J-1). This array must be recomputed at the start of each generation.

together they can be referenced using the same register. After defining the variable LEN other variables are computed based on LEN. N1 through N4 are the four locations in array SQUARES used to compute the neighborhood E value. There are several macros that make the program easier to write and read. Since the DEPTH for the program is 1, we only need one bitplane location. The routine MEMORY reserves space for arrays U, UU, V, VV, and SQUARES.

Next the values for the KEXC array are computed. The array address is in a1, R1-KEXC in d1, and the counter V in d0. Multiply d1\*d0 then divide by VEXC; add KEXC to the result and store the result at location a1+d0. Increase d0 until it reaches VMAX (100). The array for KREC is computed in the same manner. The array address is in a2, R1-KREC in d1, and the counter V in d0. Subtract V from VMAX and multiply the result by d1; divide this result by VMAX-VREC then add KREC. Store this at location a2+d0. Keep doing this until V reaches VMAX. DEMO2 fills the left side of array U and UU with 1 to represent the excited state of the media. Each array value set to 1 is also colored on the screen.

The main part of the program is next. CONTINUE fills in array SQUARES at location (1,1) in U and SQUARES. The value at U is



# OWN AN AMIGA 1200 TWICE THE SPEED OF AN A4000/030 OR A3000 SYSTEM! GET TRUE 32-bit WIDE, ZERO WAIT-STATE FASTRAM ACCESS!

\*PURCHASE THE MICROBOTICS M1230XA CARD FOR THE AMIGA 1200\*

AMIGA	
A4000 Computer	2399
A2000 Computer	599
A1200 Computer	CALL
8800 Computer	349
1942 Multitouch Monitor	489
2024 Monochrome Mon.	189
1084S Monitor	249
A2630 Accelerator	375
A320 Video Adapter	34
A2088 XT Bridgecard	49
A2091 Hard Drive & Ram	
Controller w/120MB HD	279
A2091 HD Controller	69
2 MB Ram For 2091	80
HD Floppy Drive 1.78MB	99.85
External Version	125
Janus 2.1 Update	35
A2000/3000 Keyboard	59.95
A2000/3000 Power Supply	109

**A2320 Flicker Fixer**  
Only \$129

AMIGA CUSTOM CHIPS	
Kickstart 2.1 Upgrade Kit	77.95
2.04 Rom (HiD Drives)	33.95
Kickstart 1.3	21.95
1MB Agnus (8372A)	37.95
2MB Agnus (8372B)	59.95
Super Denise (8373)	28.95
Paula (8364) Or Denise	18.95
CIA (8520)	9.50
Gary (5719)	13.95
2820/2630 Upgrade Kit	35
2091 Upgrade Eeprom 7+	35

IVS	
Grand Slam	229
Grand Slam 500	267
Trumpcard Pro	139
Trumpcard 500 Pro	225
Trumpcard 500 Plus	149
Trumpcard 500 AT	164
Printer/Interface Auxiliary	
Printer Port	55
Source Switching	
Power Supply	99

**GVP 68030/  
68882 - 25 Mhz  
1MB/60ns RAM  
w/SCSI Controller**  
**\$395**

CALL OUR BULLETIN BOARD  
THE WORLD'S LARGEST  
AMIGA RUN 6891  
10 HST D/S LINES  
2 GIGABYTES OF STORAGE  
(302) 836-6175

DKB	
Inleider II w/OK	159
2632 w/4Megabytes	399
MegAChip 2000/500	
w/2MB Agnus	169
Multi-Start 2 Rev 6A	29
KwikStart II for A1000	89
SecureKey Security Board	99
8ah Disk battery backed static RAM disk	199

## MICROBOTICS

### M1230XA ACCELERATOR

\*68030 RC25Mhz CPU w/MMU\*

- 68882 Math Chip + \$75  
Own An A1200 TWICE The Speed  
Of The Amy 4000/030! Only \$395!!

**Only \$299**

### MBX 1200z COMBO BOARD

68881 RC20Mhz FPU & w/o clock

w/1MB RAM \$139  
w/2MB RAM \$179  
w/4MB RAM \$279  
w/8MB RAM \$479

**Only \$129**

\*Add \$20 For Clock & Battery\*

Call for other custom configurations

GVP	
A500-HD8+0MB/40	275
A500-HD8+0MB/80	329
A500-HD8+0MB/120	379
A500-HD8+0MB/213	499
A530-HD8+1/120	555
A530-HD8+1/245	599
A2000-HC8+0MB	145
SIMM32/1MB/60ns	69.95
SIMM32/4MB/60ns	169.95
1MB SIMM GForce A3000	Call
G-Link Genlock	385
A3000-Impact Vision 24	1199
A2000-IV24 Adapter	45
VIU-CT	499
A1230 Turbo+ 40/40/4	\$665.00
A1290 SCSI / RAM+ LOWEST	
GForce Accelerators w/ 68030, 68882, 40ns RAM, SCSI Controller, RAM Card in One	
40Mhz/4MB	675
50Mhz/4MB	899
68040 33Mhz A2000/4MB	999
68040 28Mhz A3000/2MB (40ns RAM)	975
PC286 Module 16Mhz	59
Tahiti-II 1GB (35ms)	3300
Tahiti-II 1GB Cartridge	299
Syquest 44MB Removable	269
44MB Cartridge	65
Syquest 88MB Removable	365
88MB Cartridge	119
68882 40Mhz FPU PLCC	129
FlashROM Kit (For HDs)	35
Cinemorph Software	51
Phonepak VFX	239
DSS8 Sound Sampler	49
VO Extender (2SerialPort)	99
Image F/X	189

**Video Toaster  
Systems  
3899.00**  
25 Mhz w/6MB Ram & 120HD  
& DPS TBC III & GVP Combo

**VECTOR** THIRD GENERATION 68030  
PROCESSOR ACCELERATOR  
FOR THE AMIGA 2091  
**\$559**



ASK ABOUT OUR ACCELERATOR, HARDDRIVE AND MEMORY UPGRADES

### DERRINGER ACCELERATORS

**Only \$399**

Running at 25Mhz with MMU  
1MB 32bit RAM Exp. to 32MB  
w/ 68881 \$399  
w/4MB Ram \$499  
w/8MB Ram \$699  
w/ 68882 FPU add \$75  
w/ 68882-50Mhz add \$135.95

**50 MHZ VERSION  
Now Available  
Only \$699**

50Mhz CPU (w/MMU) & 4 MB 60NS Ram

### HARD DRIVES

Warranty	
Quantum-2 Years	
Maxtor-1 Year	
Maxtor 120 LPS 15 MS	215
Maxtor 213 LPS 15MS	299
Quantum 85 ELS	176
Quantum 105 PDrive	199
Quantum 240 LPS	365
Quantum 525 LPS 10MS	899
Quantum 1.2 Gig	1399

### A1200/600 HD's

60MB Western Digital	\$175
85M Conner	\$225
128M Maxtor	\$325
235MB Seagate	\$469

### MEMORY CHIPS

1VS 1MB SIMMS	29.95
1x8 100-80ns SIMMS	30.00
4x8 80-80ns SIMMS	119.00
1x4 80-80ns Static ZIP	17.50
1x4 80-80ns Page ZIP	15.85
1x4 80-80ns Page DIP	15.85
1x1 120-70ns DIP	3.25
256X4 120-60ns DIP	4.00
256X4 120-60ns ZIP	4.00
A4000 SIMMS 80-80NS	139.00

MATH CHIPS, CPU's & FPU's	
68030-RC-50 w/MMU	179.00
68882-RC-50 (PGA)	138.88
68030-RC-33 w/MMU	148.00
68882-RC-33 (PGA)	85.00
68030-RC-25 w/MMU	105.00
68882-RC-25 (PGA)	75.00
68030-FN-PLCC (Various)	CALL
68882-FN-PLCC (Various)	CALL
68030-256X (Bridgecard)	69.95
Crystal Oscillators (All)	10.00

### LASER PRINTER MEMORY

HP II, IID, IIP, III, IIID, IIP AND ALL PLUS SERIES	
Board with 2MB	79.00
Board with 4MB	145.00
Deskjet 256K Upgrade	59.95
HP 4 (4 Meg)	149
HP 4 (8 Meg)	295

A500/1200 Accesories	
PCMCIA 2 and 4MB	129
Baseboard 601	359

### US ROBOTICS

16.8K Courier HST	
with fax	549
16.8K Courier HST Dual	
Standard with fax	749
Courier v.32bis	449

**14.4K Sportster  
With or W/O FAX  
\$199**  
Add \$40 For Fax Version

### ACCESORIES/MISC.

Kick Back ROM Switcher	35
PowerPlayers Joystick	6.49
CSA Rocket Launcher	499
SupraTurbo 28Mhz	145
Safeskin Protectors	15.00
Xirastor+ Chip Puller	9.95
Shinghol single A2000 slot for the A500	39.95
Kool-It Cooling kit A500	39.95
QwikA Syntha 4 socketed ROM selector	39.95
Power Connectors	CALL
SCSI HD Cable	CALL



18 Wellington Drive  
Newark, DE. 19702  
(800) 578-7617 ORDERS ONLY  
(302) 836-4138 PRODUCT Info/Tech  
(302) 836-8829 Fax 24 HOURS

### Please Read Our Policies

Visa / Master Card Accepted. Prices And Specifications Are Subject  
To Change Without Notice 15% Restocking Fee On All Non-Defective  
Returned Merchandise. Call For Approval RMA! Before Returning  
Merchandise. Shipping And Handling For Chips Is \$5 COD Fee \$5  
Personal Checks Require 10 Days To Clear. Call For Actual Shipping  
Prices On All Other Items. If You Don't See It Here, Call Us!

added to the value just to the left in SQUARES (-1(a5)), just above (-LEN(a5)), and subtracted from the value to the left and above (-LENPI(a5)). This continues until the whole array is filled. Figure out the initial value for yourself for this array and it becomes more clear as to how it works when you compute neighborhood values. The values get large enough that they could be stored in a word array, but since we're only using their difference, we can get by with a byte array.

## TESTING THE CELLS

The TEST routine has to start at (R+1,R+1) in all the arrays since two of the locations used in SQUARES are R-1 away from the current cell. The current V value is put in NEWV and saved in TESTV; the current U value is stored in U. If the current U is 1, branch to TEST1. If it is 0, however, first compare TESTV to VEXC. If it's lower, branch to TEST0A. If not, compute the neighborhood using N1+N2-N3-N4 in array SQUARES. Compare this value with the corresponding value in the KEXC array; if it's less or the same, branch to TEST0A else change UU to 1.

TEST0A subtracts GDOWN from TESTV. If there is a carry the result went below 0 so replace TESTV with 0. Store TESTV in VV.

TEST1 follows the same general procedure. Compare TESTV to VREC; if it's lower or the same, branch to TEST1A. If not, compare TESTV to VMAX (100). If they're equal, branch to TEST1B. If not, first compute the neighborhood E value using N1+N2-N3-N4 in array SQUARES. Since this time we want the number of unexcited U cells, subtract the E value from RR. Compare this to the corresponding value in array KREC; if it's greater, branch to TEST1B.

TEST1A adds GUP to TESTV. Compare the result with VMAX; if it's greater, make the result VMAX. Store TESTV in array VV. TEST1B reverses the U value to 0 and stores this in array UU.

When an entire row in an array has been computed you have to jump 2R+1 (R0) to the next location since the row stops R away from the end and starts at R+1. After the entire array has been checked the new UU and VV values are switched back to U and V while the corresponding SQUARES locations are cleared in preparation for re-filling them. The PSET routine takes the value of each UU (either 0 or 1) and rotates it into d3; when d3 is filled it's poked into the proper screen location. When a row has been displayed the next row is drawn 32-bits at a time through the length of the array. When you press the LMB the program terminates else it reverts back to CONTINUE and re-starts by filling the array SQUARES.

Type or copy this program from the magazine disk as RECOVERY.ASM. Use the enclosed A68K to assemble it by typing A68K RECOVERY.ASM; then use BLINK to link the program by typing BLINK RECOVERY.O. Run the program as RECOVERY. I usually run this program from RAM; since the disk drives sometimes have a tendency to keep running while the program is running. There are alternate values you can try at the end of the listing. If you want to experiment with your own values, be sure to meet the following requirements:

```
0 < VEXC < VMAX < 100
```

```
0 < GDOWN < GUP < VEXC
0 <= VEXC < KREC <= R12h+1
KREC(0) < 255 < KREC(100) < 255
```

The last two requirements are artificial restrictions since I'm using only byte arrays and values can't exceed 255. I've included CHART 1 recapping the test rules depending on the U value.

## CHANGES

Some changes you might want to try with this program are to assign different lengths and widths making a rectangle instead of a square (use LENX and LENY). Or color the V values instead of the U values; this will require that you expand the PSET routine for up to 5-bitplanes. Finally, you might want the program to accept the variables from CLI or "read" the DATA lines at the end of the program.

## Listing

```
; LISTING1
equates:
depth          = $,
jam2           = $1
mousebuttons   = $8
borderless     = -$800
wv.screen      = $2e
wv.rport       = $32
nv.screen      = $1e
customscreen   = $f
activate       = $1000
rmb            = $1000
public         = $1
chip           = $2
fast           = $4
clear          = $10000

offsets:
;excc
openlibrary    = -552
closelibrary   = -414
allocmem       = -198
freemem        = -210
forbid         = -132      ;no registers used
permat         = -138      ;no registers used
;intuition
openscreen     = -198
closescreen    = 66
openwindow     = -204
closewindow    = -72
viewportaddress = -300 ;a0=window
;graphics
setdrmd        = -354
loadrgb4       = -192      ;a0 vp,a1-colortable,d0=pens
setapen        = -342
```

writepixel 24

```
sum      equi d'
resrv    equi d'
c        equi d4
across    equi d6
down      equi d5
len      = 32*4 ;try 32*4 for AMIGA 3000
lenm1 = len-1
mlenm1 = -1*lenm1
lenm2 = len-2
xoff = (320-len)/2 ;to center display
yoff = (200-len)/2 ;to center display
lenp1 = len-1
mlenp1 = -1*lenp1
lenm3 = len-3
mlen = -1*len
byte = (yoff*320+xoff)/8 ;byte containing yoff
wpl = len/32-1 ;long words per line
byteoff = 2*xoff/8 ;offset to next byte
size = len*len ;array size
:
:
rpl = r+1
n1 = r*lenp1
n2 = (r+1)*mlenp1
n3 = r*lenm1+1
n4 = 1*(r+1)*len+r
r0 = 2*r+1
r1 = r*(2*r+1)
r2 = (2*r+1)*(2*r+1)
vmux = 100
vréc = 70
vexc = 05
gup = 20
qdown = 5
krec = 5
kexc = 1
```

```
macros:
syslib macro ;(routine)
movea.l #4,a6
jsr 1(a6)
endm
```

```
openlib macro ;(name,location,BEQ if 0)
lea 1,a1
moveq #0,d0
syslib openlibrary
move.l d0,\2
beq \3
endm
```

```
intlib macro ;(routine)
movea.l intbase,a6
jsr 1(a6)
endm
```

```
openscreen macro ;(parameters,screen,BEQ=0)
lea 1,a0
intlib openscreen
move.l d0,\2
beq \3
endm
```

```
openwindow macro ;(parameters>window,BEQ=0)
lea 1,a0
move.l screen.nw,screen(a0)
intlib openwindow
move.l d0,\2
beq \3
endm
```

```
gfxlib macro ;(routine)
movea.l gfxbase,a6
jsr 1(a6)
endm
```

```
pset macro
move.l rp,a1
move.w across,d0
add.w #xoff,d0 ;center across
move.w down,d1
add.w #yoff,d1 ;center down
ext.l d0
ext.l d1
gfxlib writepixel
endm
```

```
color macro
move.l rp,a1
gfxlib setapen
endm
```

```
array macro ;(address,BEQ=0)
move.l #size,d0
move.l #$10004,d1
syslib allocmem
move.l d0,\1
beq \2
endm
```

```
free macro ;(array name)
move.l #size,d0
movea.l 1,a1
syslib freemem
endm
```

```
evenpc macro
ds.w 0
endm
```

```
start:
move.l sp,stack
openlibs:
openlib intuition,intbase,done
openlib graphics,gfxbase,close_int
```

```
set_up:
make_screen:
openscreen myscreen,screen,close_libs
openwindow mywindow>window,close_screen
move.l d0,a0
movea.l ww.rpart(a0),a1
move.l a1,rp
movea.l window,a0
intlib viewportaddress
move.l d0,vp ;viewport address
```

# The LANGUAGE For The Amiga!

**T**his new package represents the fourth major upgraded release of F-Basic since 1988. Packed with new features, 5.0 is the fastest and fullest yet. The power of C with the friendliness of BASIC. Compatibility with all Amiga platforms through the 4000... compiled assembly object code with incredible execution times... features from all modern languages, an AREXX port, PAL and ECS/AGA chip set support... Free technical support... This is the FAST one you've read so much about!

## F-BASIC 5.0™

Supports DOS  
1.3, 2.0, 2.1 and 3.0

**F-BASIC 5.0™ System \$99.95**

Includes Compiler, Linker, Integrated Editor Environment, User's Manual, & Sample Programs Disk.

**F-BASIC 5.0™ + SLDB System \$159.95**

As above with Complete Source Level Debugger.

Available Only From: DELPHINOTIC SYSTEMS, INC (605) 348-0791

P.O. Box 7722 Rapid City SD 57709-7722

Send Check or Money Order or Write For Info Call With Credit Card or C.O.D.

Fax (605) 342-2247 Overseas Distributor Inquiries Welcome

```
movea.l rp,a1
move.l #1a2,d0
qtxlib setdrnd
get_bit_planes:
movea.l rp,a1
movea.l 4(a1),a1
lea bp1,a0
move.l 8(a1),a0 ;bitplane addresses
syslib forbid
memory:
array u,close_window
array uu,close_outu
array v,close_outuu
array vv,close_outv
array squares,close_outvv

lea akexc,a1
moveq #0,d0
moveq #0,d1
moveq #0,d2
11 move.b #r1,d1
sub.b #kexc,d1
mulu.w d0,d1
move.b #vexc,d2
divu.w d2,d1
add.b #kexc,d1
move.b d1,0(a1,d0)
```

```
addq.w #1,d0
cmp.b #vmax,d0
bls.s 11

lea akrec,a2
moveq #0,d0
moveq #0,d1
moveq #0,d2
12 move.b #r1,d1
sub.b #krec,d1
move.b #vmax,d2
sub.b d0,d2
mulu.w d1,d2
move.b #vmax,d1
sub.b #vrec,d1
divu.w d1,d2
add.b #krec,d2
move.b d2,0(a2,d0)
adrq.w #1,d0
cmp.b #vmax,d0
bls.s 12
```

```
demo2:
movea.l u,a4
movea.l uu,a5
moveq #0,down
demo22
moveq #0,across
demo21
move.b #1,(a4)+
move.b #1,(a5)+
moveq #1,d0
color
pser
addq.w #1,across
cmp.w #len/2,across
bne.s demo2!
lea len/2(a4),a4
lea len/2(a5),a5
addq.w #1,down
cmp.w #len,down
bne.s demo22
```

```
continue:
movea.l u,a1
lea lenpl(a1),a1
movea.l squares,a5
lea lenpl(a5),a5
moveq #1,down
cont2
moveq #1,across
cont1
move.b (a1)+,sum
add.l 1(a5),sum
add.b -len(a5),sum
sub.b -lenpl(a5),sum
move.b sum,(a5)+
addq.w #1,across
cmp.w #len,across
bne.s cont1
lea 1(a1),a1
lea 1(a5),a5
addq.w #1,down
```



## MOVING?



## SUBSCRIPTION PROBLEMS?

Please don't forget to let us know. If you are having a problem with your subscription or if you are planning to move, please write to:

**Amazing Computing Subscription Questions**  
**PiM Publications, Inc.**  
**P.O. Box 2140**  
**Fall River, MA 02722**

Please remember, we cannot mail your magazine if we do not know where you are.

Please allow four to six weeks for processing.

```

n_solve
  free v
  close_stdout
  free u
  close_stdout
  free u
  close_stdout
  movea.l window, a
  intlib:closewindow
  close_screen:
  movea.l screen, a0
  intlib:close_screen
  close_libs:
  movea.l gfixbase, a1
  liblib:closelibrary
  se_int:
  movea.l intbase, a1
  liblib:closelibrary
done:
  movea.l stack, sp
  rts

evenpc

stack dc.l 0
gfixbase dc.l 0
intbase dc.l 0
screen dc.l 0
window dc.l 0

```

```

rt dc.l 0
vp dc.l 0
u dc.l 0
v dc.l 0
vv dc.l 0
square dc.l 0
bp1 dc.l 0
bp2 dc.l 0
bp3 dc.l 0
bp4 dc.l 0
bp5 dc.l 0
ikex dc.l b 102
  evenpc
akre dc.l b 102
  evenpc

graphics dc.l 'graphics.library',
  evenpc
intuition dc.l b 'intuition.library',
  evenpc

myscreen
  dc.w 0,0,320,200,depth
  dc.b 0,1
  dc.w 0
  dc.w customscreen
  dc.l 0,0,0,0
  evenpc

mywindow
  dc.w 0,0,320,200
  dc.b 0,1
  dc.l mousebutton
  dc.l borderless|activate|mb
  dc.l 0,0
  dc.l 0
  dc.l 0,0
  dc.w 0,0,0,0
  dc.w customscreen
  evenpc

data:
  dc.b '1,1,65,0,5,5,1'
  evenpc
  dc.b '3,85,89,23,19,7,5'
  evenpc
  dc.b '6,82,71,15,5,53,0'
  end

```



**Please write to:**  
**William Nee**  
**c/o AC's TECH**  
**P.O. Box 2140**  
**Fall River, MA 02722**



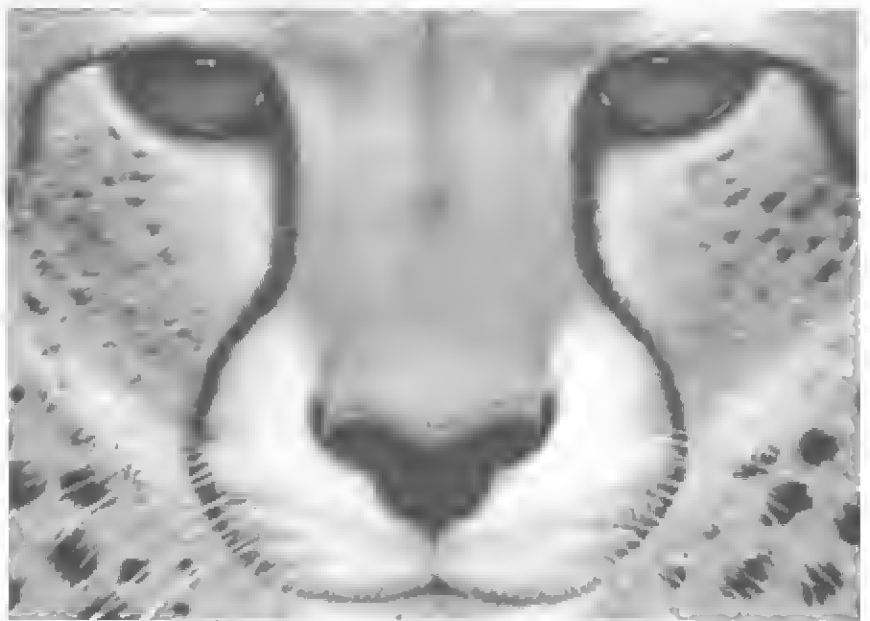
*W*hat is morphing exactly? Why is it so popular, right now? Is it that complicated? How can it be implemented? Which are the best techniques, and why? Why do they usually require so much user intervention? In this article, we're going to try to answer these questions and make you see morphing as a technical matter, instead of an unexplainable magical effect. Well, let's say that the results may look magical sometimes, but the technique surely is not.

You won't find the word morphing in any dictionary, as it has been just recently created to describe a new special effect that you should have seen a couple of times by now. The word comes from a better-sounding, shorter term of the more complicated synonym *metamorphosis*, which is simply a change of shape, structure, or substance of something into another. This kind of transformation is many times a desirable special effect, especially in the entertainment and advertising industries. You have probably seen many metamorphoses in movies and commercials, even before computer based morphing techniques showed up. Conventional horror movies, for instance, are filled with transformations, which have been traditionally created using a couple of elementary techniques, such as rudimentary scene cuts, mechanical models that can change their appearance, and cross-dissolves.

*All you ever  
wanted to  
know about  
morphing\**

*by*

*Bruno Costa & Lucia Darsa*



*\*but were afraid to ask...*

Cross-dissolve has been, by far, the most-used method to create metamorphoses, probably because of its inherent simplicity and the acceptable end results. Since the initial and final subjects in a transformation usually differ significantly, some dolls representing in-between steps in the transformation have to be created. The cross-dissolve is then performed between each pair of successive dolls, which must be positioned so that they match reasonably well. The obvious problem with this technique is that the dolls never match perfectly, and that the transformation looks as though it is "broken" into many smaller transformations, from each doll to the next. Besides, the creation of convincing in-between dolls may be a hard job, although we are often surprised by what special effects technicians can do.

effective than a completely synthetic 3-D metamorphosis sequence. Obviously, these crossdissolves can also be created with computers instead of optical techniques and lenses. At first, there is no good reason to do that, since optical cross-dissolves are perfectly effective, but computers are able to do much more than lenses—including simulating them!—so they might present a solution to the alignment problems that plain cross-dissolves have. It is not hard to realize that a much better effect can be created if the images to be transformed are geometrically aligned before doing the cross-dissolve, so that the transition is smooth and fluid. This artificial alignment between the images is basically the only (but great) difference between a traditional cross-dissolve and the so-called morphing effect.



1a  
1b, page 19

Different approaches, based on computer graphics, can be used to simulate metamorphosis-like effects. Two 3-D models, if they have a similar structure (same number of vertices, etc.), can be trivially transformed into each other by interpolating the position of corresponding points (see the "Interpolation" box). With extra effort, some objects with highly differing structures can also be transformed. However, even with the corresponding interpolation of surface characteristics like color and texture, the final quality of a 3-D object metamorphosis will depend on the accuracy of the models and the realism of the rendering. For real world objects such as live beings—typical metamorphosis subjects—convincing modeling and rendering are currently impractical, if not impossible.

When carefully designed, a basic cross-dissolve, even if the intermediate steps do not match perfectly, is usually much more

It was not until recently that the researchers working for top computer graphics companies found out that such techniques could be used to create much better metamorphoses using computers and digital images. The *but* of this effect was seen in the movie *Willow*, where a goat turns into many different animals and then finally into a sorceress. Since then, morphing has been significantly improved, and used in many different movies, sometimes in unsuspecting places. In *Indiana Jones and the Last Crusade*, for instance, traditional techniques were used to create parts of the transformation where the villain ages rapidly until death, and morphing was used to join the different transformation segments smoothly. Morphing has also been used explicitly in the anthological clip *Black or White*, one of the best

examples of high quality morphing of moving subjects, and in *Terminator 2*, for most of the transformations of the shape-shifting creature into human beings. Morphing is now such an important tool in the special effects industry that most movies with a reasonable budget for this area sports at least one morphing sequence, and this includes productions like *The Abyss*, *Star Trek VI*, *Freddy's Deal*, *The Final Nightmare*, *Bram Stoker's Dracula*, and so on.

Now, with the availability of capable morphing software for the Amiga, you should expect to see morphing incorporated into the toolbox of lower budget special effects producers. Even though some of these packages do not have the features that are needed to create truly professional results (features which we're going to detail later), it is possible to use most of them to create simple metamorphoses, suitable for TV commercials and low-budget TV productions. More complex packages, if used with high quality display hardware like a film recorder, can be used to produce quite affordable but convincing special effects for movies.

### Morph = Warp + Dissolve

We'll now see, through a simple example, how the alignment of the main image features before doing a cross-dissolve can produce much better results than cross-dissolve alone. Figures 1a and 1b show two digitized images, in their original, undeformed states. Let's call Figure 1a the FROM image and 1b the TO image. Figure 1c is a 50% cross-dissolve between 1a and 1b. Notice how awful the result can be if the eyes, nose, and mouth do not match: it definitely doesn't look like an intermediate step of a metamorphosis.

What if we try to move the woman's eyes a bit apart, or move the feline's mouth up so that it matches the woman's mouth? By pushing or pulling the images here and there, just as if they were made of rubber, we can deform (or warp) them somehow, so that their eyes, nose and mouth match. Figures 1d and 1e are deformed versions of Figures 1a and 1b. Note how the main features, like the mouth, are now at the same approximate position and shape in both pictures. Figure 1f shows a similar 50% cross-dissolve of Figures 1d and 1e, and now the results are a lot better (the quality of the morphing can be enhanced even further with the use of advanced techniques that we'll explain later). You might notice that although the deformed images usually look somewhat weird and artificial, their combinations produce creatures that are very authentic and believable, almost lifelike.

Obtaining this kind of alignment is not so straightforward, as it requires that the main features of both images are identified and then correlated, so that an arbitrary warping of the images can be done. Although the identification of the main features could be done by an edge-detection algorithm, for instance, there is no known way to automatically correlate corresponding features in both images. Presently, all this work must be done by the animator who operates the software.

There are a couple of different ways of specifying these features, but the important point is that the animator's work must give enough information for a particular warping algorithm to align the features. Once such a warping algorithm is used, the cross-dissolve is simply a pixel-by-pixel weighted average of the two deformed images.

## Interpolation

You'll find the word interpolation many times when working with morphing. In fact, we could say that plain morphing is just a sequence of interpolations (two complex shape interpolations followed by one color interpolation, plus interpolations in time for animation morphing).

An interpolation is simply a combination of two values, with the quantity of both controlled by a single parameter. This parameter can be given as a percentage: 0% means just the first value, 100% means just the second value, and in-between percentages represent values between the first and the second. The simplest type of interpolation is the linear interpolation, which is simply a weighted average between the two values, with the weight as the parameter. A linear interpolation between 20 and 50 is represented in the table below for some particular percentages (of course there are infinitely many).

You are not restricted to linear interpolations, though, there are many ways of going from one value to the other. In general, an interpolation will be a weighted average between the two values, where the weight is a function of the percentage.



le

A simple fact that is sometimes confusing is that morphing is an operation performed on two different images and that warping operates on a single image. In fact, two warps are necessary to create a morph: each of the two images to be morphed is warped independently, and then the results are combined with a cross-dissolve. As mentioned before, when you are creating a morph, you'll want to warp each of the images so that its features match those of the other. To do that, of course, you'll have to know where the features of both images are, so the warps are not fully independent in fact. The whole morphing process is depicted in the small diagram in Figure 1.

### Animation Morphing

When it comes to morphing, the word "animation" can appear in two different contexts: either describing the output of a morphing software or its input. We've already seen how an image representing one of the intermediate steps between two other images is created. It is easy to see that there are infinitely many steps between two images, ranging from steps which are "almost" the first image to steps which are "almost" the second. We like to identify each of these steps through a percentage, so that 0% would be exactly the first image and a 100% would be the second one. Any percentage can be used, and of course we can vary this value continuously from 0 to 100, so that a coherent sequence of images (an animation!) is produced. This is not what is called animation morphing, though; it's simply an animation created from two still images.

An animation morphing is created between two moving subjects, instead of two static images. This is the same as saying that two animations are used as input to the morphing software. The difference from still to animation morphing is striking, it brings a whole new level of realism in the results. There is no better example of this than the *Black or White* video, where people dance and sing while morphing takes place.

Do not be fooled that animation morphing is easy to do: it takes a good deal of work to find (or create) two animations that match reasonably well, so that a convincing morph can be created. This difficulty is the reason why there are so few examples of animation morphing, even in movies with a reasonable budget. In some of those productions it is even possible to see a moving subject stop, metamorphose, and then continue its movement, just to be able to use a simpler still morph instead of a full animation morph. In the multi-million-dollar movie industry, things like these might bring a saving of some hundred thousand dollars! Technically speaking, both kinds of morphing have similar complexities, but animation morphing requires a lot of extra effort from the animator.

You shouldn't be surprised to know that an animation morph isn't much more than a combination of many still morphs. As mentioned before, two animations are used as input to the morph, both with the same number of frames  $n$ . Between each pair of corresponding still frames from the two animations, it would be possible to create an entire still morph with  $n$  frames, which are



1c

depicted in the figure as empty rectangles. Of all these  $m \times n$  possible frames, we pick up just the  $n$  highlighted ones to be rendered and placed in the final output animation, and, believe us, it works! The reason for that is simple: for each successive frame, we morph a bit toward the TO animation and move a bit toward the end of the animations. Note also that we start from the first frame of the FROM animation, and end precisely at the last frame of the TO animation. It is a simple concept, but it takes a while before you get used to it.

### Warping Algorithms

Since cross-dissolve is so simple, it is the warping phase that will determine the speed, quality, and even the user-interface of a morphing software. There are basically two techniques for specifying the important features of an image, and thus, how it will be warped.

The first technique to appear was based on control points placed over each important feature of the image. Each of these control points has a corresponding one which can be moved to show the deformation of the region of the image that surrounds that control point. When used for morphing, these corresponding control points are placed over the other image to specify the corresponding features. Since each feature is usually much larger than a single point, many points must be placed over each feature to produce acceptable results.

The other technique is based on two regular meshes. The first is

adjusted so that it follows the overall shape of the underlying image, and the second is a deformed version of the first. Both meshes have exactly the same structure and number of points. The meshes can be formed either by straight lines connecting each of the adjacent mesh points, or by any smooth spline curve (see the "Splines" box) passing over each line or column of points. When used for morphing, the second mesh is adjusted so that it follows the shape of the other image, and corresponding points of the two meshes are placed over corresponding features.

Once the animator has placed the "feature indicators," either single points or a mesh, over each image, a suitable algorithm must be used to compute the deformation. The algorithm used when features are specified by single points is relatively simple to implement. Basically, a triangulation, or a set of triangles, connecting all the control points in the deformed position is automatically created by the program. For each pixel in the output image, we must find which is the triangle that covers it. We then look at the original triangle (formed by the same three vertices in their undeformed position, in that case A, B and C), and find the position (p) in this triangle where our pixel would be. The color of the original image at this position will be the color of the output pixel. The entire procedure is repeated for each pixel in the output image.

Although the algorithm above is easy to implement and relatively efficient, the quality of the results is not what one would call protes-

sional. If you use too few triangles for a given image, discontinuities of the deformation will be noticed exactly along the edges of the triangles. Of course there are workarounds that try to smooth the deformation around the triangle edges, at the cost of speed. Note, however, that since the triangles are arbitrarily created by the algorithm, the position of the edges is unpredictable, and it may change suddenly during a continuous morph. If you have access to a program that uses point-based techniques, you can easily check the existence of these discontinuities—they will be noticeable if you try to warp a large checkerboard pattern. All these problems will be solved by an appropriate implementation of a smooth spline-mesh warping algorithm.

### Spline Mesh Warping

The implementation of a spline-based warping technique that we present here is considerably more complex than the point based one, but it has many advantages. First, it is naturally continuous and smooth, providing for much more fluid and spontaneous deformations, specially for living subjects. Second, it is an efficient implementation that works in two passes, facilitating anti-aliasing, and it takes advantage of the memory organization of conventional computers. Besides, although the algorithm itself is more complex, the required data structures for a full implementation are simpler and easier to deal with.

The efficiency of the algorithm is mostly due to its two-pass nature: it can be performed in two independent passes, one horizontal, that produces an intermediate image, and one vertical, that uses this intermediate image to produce the final output. This brings the

problem of deforming a 2-D image down to the simpler problem of deforming many 1-D scanlines, greatly reducing the required calculations. Let's see how this can be done.

The horizontal pass of the algorithm starts by creating an intermediate mesh *I* that includes just the horizontal displacements from the source mesh *S* to the destination mesh *D*, that is, each point in *I* has the same *x* coordinate of the corresponding point in *D*, and the *y* coordinate of the point in *S*. Each column of *S* and *I* is then fitted with an interpolating spline—we call *SS* the set of vertical splines in *S*, and *IS* the set of vertical splines in *I*. Each horizontal scanline is intercepted independently with *IS* and the *x* coordinates of the interceptions with *SS* are then fitted with a new interpolating spline, which gives a smooth relation, called the scanline mapping function, between each pixel in the output and the pixels in the corresponding input scanline. For each pixel of each horizontal scanline of the intermediate image, it is now easy to use this mapping function to determine which pixels of the input image influence this output pixel. The color of pixel *p* will then be either the color of *a*, that of *b*, or an average of the two to obtain an anti-aliasing effect. In general, anti-aliasing here is as simple as computing an average of the pixels that affect a particular output pixel, assigning higher weights for the pixels that have greater influence, as in this case.

You should note at this point that there are two ways to apply a mapping function to an image: direct and inverse mapping. In the first case, the function is applied to each pixel in the input image to find its

## Splines

Splines are curves that are defined by a small set of control points, and a (usually complex) formula that takes these points as arguments. With the use of this formula, it becomes possible to know all the infinitely many points that make up the curve without having to store all of them. A spline will usually pass on, or near, each of its control points, and the splines that pass exactly over each of the control points are called *interpolating splines*.

There are many different kinds of splines, each of them defined by a certain formula: Natural Splines, Beta Splines, B-Splines, NURBS, etc. You can find the formulas and explanations for most of them, and some other, in good introductory computer graphics books like the second edition of "Computer Graphics: Principles and Practice" by J. Foley, A. van Dam, S. Feiner and J. Hughes.



location in the output. Inverse mapping, on the other hand, applies the inverse function to each pixel in the output to determine which input pixels contribute to it. Inverse mapping is generally preferred since it assures that every output pixel is calculated and that no work is wasted computing unnecessary pixels, as opposed to direct mapping.

The vertical pass uses the intermediate mesh and the intermediate image as input to produce the final output; the source mesh and the source image are not used. It is completely analogous to the first pass, with the rows of I and D fitted by horizontal splines, and the mapping functions computed for each vertical scanline. Note that the intermediate image may be distorted to such an extent that there is not enough information to compute a correct vertical pass, but this rarely happens, unless there are very large rotations in the two meshes, but this is usually nonsense for morphing.

It may seem complicated to implement such an algorithm, but once you have the routines to manipulate the kind of spline that you chose, it is not that hard. Obviously, this is just the basic warping algorithm, it needs a bit of extra work if you want it to do morphing. For this, it would need to warp two different images according to their respective meshes, and then create a cross-dissolve between the results, an average between each pair of corresponding pixels. By the way, the images must have exactly the same dimensions for this to be possible!

### Progress Control

An important feature missing from the above discussion is the detailed control of the progress of the transformation. There are two

different things going on in a morph, the warping and the cross-dissolve—OK, we know you know that by now—so you should be able to control the progress of each of them independently. At a certain point in an animation, for instance, you might want to have the result almost fully warped to the end shape, but with the colors of the starting image still visible. You can also have no warping at all, keeping it at zero, and let just the cross-dissolve take place, or vice-versa. In more conventional morphs, you'll want both of them to take place, but at different rates, usually starting with pure warping and letting the cross-dissolve in just at a later stage. The progress of each of the two can be precisely described by two functions, which we like to call shape mixer and color mixer, that control how much of the FROM and the TO images will show up in the output, or the mix between the two.

Even more important than the overall control of the color and the shape progresses is the ability to define them precisely at specific regions of the image. This is achieved through the use of local mixers. These mixers are created and manipulated just like the global ones, but they can be associated with particular control points in the mesh, and their action is restricted to the region that surrounds them. Of course you can have all kinds of mixers working at the same time, with global ones controlling the rates for most places, and local ones providing distinctive rates for some special parts of the image. The ability to do this is an absolute requisite to obtain professional results like those you see in the movies and TV commercials.





Id

## Conclusion

You would be surprised to know that the code directly devoted to morphing in a serious morphing package will hardly take up more than 15% of the total. Video professionals require much more from the morphing software they are going to use, like a friendly user-interface, many types of image conversion routines, dithering, anti-aliasing, alpha-channels, high-quality output, multi-format image loading and saving, and an efficient and system-compliant implementation. They also need support for creating complex animations easily, and automatically recording them to video tape. In this article, we were not worried about how all of this can be done—after all, it doesn't have much to do with morphing itself.

Considering the availability of morphing packages for the Amiga, you might ask yourself why you would worry about trying to implement the algorithms we've described here. Implementing something is always an excellent challenge, and these algorithms are quite interesting; you can learn a lot from them. And much more important than that, understanding how morphing works inside will surely help you to use it much better from the outside. If you wish, you can try the demo version of *Visionaire* included in the AC's TECH disk, which will let you experiment with many of the ideas and the algorithms presented here, including spline meshes and all kinds of mixers. We hope to have helped to make some concepts which seemed alien to you become friendly and natural.

Bruno Costa and Lucia Darsa are computer engineers working with computer graphics at Rio de Janeiro, Brazil. They are the authors of *Visionaire*, the morphing software from Impulse, Inc.

A demo of an earlier version of *Visionaire* is included on the AC's TECH disk.

Please write to  
Bruno Costa & Lucia Darsa  
c/o AC's TECH  
P.O. Box 2140  
Fall River, MA 02722



# Have Your Own Custom 3-D Graphics Package: Part I

by Laura M. Morrisson

Commercial 3-D graphics packages do photorealism superbly well, but is that all there is? If you have ever wished for something more, or just something different, from 3-D graphics, here is your chance. The programs described in this article provide the necessary 3-D graphics functions and a point of departure for your own custom code.

I developed these programs to accept 3-D trees and other plant objects produced using iterated functions systems (See Volume 3, Number 1 of this journal for my article on "Make Your Own 3-D Vegetation Objects.") The IFS vegetation objects are defined by sets of points whereas commercial programs require objects defined by polygonal faces.

Besides accepting objects defined by sets of points, the programs described here provide some extra benefits not available with commercial graphics packages. The rendering algorithm is a Z-buffer algorithm, which is much faster than ray tracing. The code is designed to work on parts of the scene independently, so the number of objects and intricacy of composition is unlimited. The guiding metaphor is not Photography but Fine Arts, implying much greater artist's interference in rendering. You can use the programs as they are, or customize the algorithms and compile custom variations.

Part I of this article describes the Z-Buffer rendering algorithm for objects defined by points and gives a z-buffer rendering program. Part I includes programs for generating boxes, shadows, and patterned grounds. It also includes several support programs for handling point-defined objects, such as changing position, manually adding points, and combining picture elements.

All the lighting algorithm code is postponed until Part II, which will explain the derivation of a lighting algorithm, and a fine arts theory of coloring for Amiga 3-D graphics.

## World Space: Points and Perspective

The context of these programs is a world space with left-handed real number coordinates. A frame in the world space where  $x: 0 \leq x < \text{WIDTH}$ , and  $0 \leq y < \text{HEIGHT}$  provides a grid of  $\text{HEIGHT} \times \text{WIDTH}$  integer positions. The frame is not the same as the Amiga screen. See Illustration 1.1(a). The programs use floating point numbers for calculation, integers for saving, and only translate to Amiga screen pixel coordinates for plotting.

Objects and lights can be anywhere in the world space. No view volume or clipping is necessary. Points that do not have projections on the frame are simply ignored when time comes to plot the Amiga screen.

The eye, which defines the point of view, must be between the frame and the objects it is viewing as shown in Illustration 1.1 (a). The frame is analogous to the retina of the eye. This eye position gives more natural vision than having the frame between the eye and the object. When the frame is between the eye and the object, the object appears smaller the nearer it is to the eye, just the opposite from natural vision, and, as the distance

between the eye and the object increases, the object increases in size toward orthogonal projection. When the eye is between the frame and the object, however, the object looms larger as the object comes nearer to the eye, and diminishes as it recedes, as in natural vision. More dramatic images are possible with the eye between the frame and the image.

The distance between eye and frame is not fixed. The relative proportions of the distances between eye and frame, and between eye and object will influence the image. When the eye is about half-way between object and frame the object's image will be about the same size as the object. If an object's image is larger than the object extra points may be needed to insure an opaque image. Notice that, as on the retina, an image is upside down and right-left reversed on the frame.

Objects consist of points which are stored as four 16-bit integers: color register number, x coordinate, y coordinate, and z coordinate. For easy handling they are best stored in numbered files of, say, 4000 points each. IFS vegetation objects will typically have from 10 to 30 such files.

## Part I includes programs for generating boxes, shadows, and patterned grounds.

An object's points are independent. They have no order and are not linked. Objects can be combined by adding files of their points, or, parts of an object can be processed separately and differently by splitting the object's files into sub-sets of points.

Objects defined by points are easy to work with. Algorithms for such objects appear to be much simpler than algorithms involving polygonal faces.

### The Two-Point Form

The two-point form of the equation of a line in space is useful for 3-D graphics having point primitives. A line in space is determined by two points, P1 and P2, on the line. Any other point, P, is on the line if the coordinates of P satisfy the equations:

$$(x-x1)/(x2-x1) = (y-y1)/(y2-y1) = (z-z1)/(z2-z1)$$

Point P(x,y,z) is on the line through P1(x1,y1,z1) and P2(x2,y2,z2) if these relations hold.

These are versatile and powerful equations worth the effort necessary to become familiar with them. They provide solutions to many 3-D graphics problems. Illustrations 1.1 and 1.2 show some of the useful points and lines. Most of the algorithms for the programs given in this article exploit the two-point form of the line in space. (Detailed

discussions of relationships of lines in space can be found in Analytic Geometry texts.)

The 3-D graphics described in this article uses a Z-buffer rendering algorithm instead of ray tracing. A Z-Buffer algorithm renders a scene by keeping track, in a 'z-buffer,' of the z value associated with each point (x,y) plotted, and, when another point comes along having a smaller z-value, replacing the previously plotted point and its z value. The algorithm automatically sorts out which object is in front of which.

The z-buffer algorithm described here uses the equation of a line through two points in space to calculate which points the eye sees. Illustration 1.1 (a) depicts, in side view, the lines of interest relating the object, the frame, and the eye. For simplicity the object is shown solid but is to be considered a cluster of points. The eye is between the object and the frame. The lines suggest how the image gets inverted onto the frame. When the line through a point of the object, and the eye, intersects the frame the object is visible if not blocked by a nearer object. The equations give the point of intersection, and, with transformation, the image of the object's point on the Amiga screen.

Equations for the z-buffer rendering algorithm can be derived from the two-point equations as follows: Points on the frame have z = 0 (by definition of the frame) so the expression involving z becomes a constant:

```
Let P1 = Pobj(Xobj,Yobj,Zobj) and
    P2 = Peye(Xeye,Yeye,Zeye), and
    P = Pframe(Xframe,Yframe,Zframe)
Then (x-x1)/(x2-x1) = (y-y1)/(y2-y1) = k1
constant.
```

Equating each of the other two terms to k1 yields equations for x and y, where x and y are on the plane z=0, that is, the frame.

$$\begin{aligned} x &= x1 + (x2-x1) * x1 = x1 + k1 * (Xeye-Yobj) * Xobj \\ y &= y1 + (y2-y1) * y1 = y1 + k1 * (Yeye-Yobj) * Yobj \\ z &= 0 = z1 \end{aligned}$$

The line through the eye and the object's point might not intersect the frame if the object is too low, too high, or too far left or right. If  $0 \leq x < WIDTH$  and  $0 \leq y < HEIGHT$  then the point's projection falls within the frame, and has a representation where the line intersects the frame. Finally, Pframe has a corresponding pixel, Pscr(Xscr,Yscr,Zscr), on the Amiga screen. Pframe can be transformed into Pscr with the equations:

$$\begin{aligned} Xscr &= WIDTH * Xframe \\ Yscr &= HEIGHT * (HEIGHT-Yframe) \end{aligned}$$

The z value = Zobj. Note that the z buffer holds a record for each screen pixel of the z value associated with that pixel.

The algorithm for calculating the shadow of an object on the ground also uses the two-point form of the equations of a line in space to find points of the shadow object. Illustration 1.1 (b) shows the relevant points and lines. The points are:

$$\begin{aligned} P1 &= P1sc(X1sc,Y1sc,Z1sc) \\ P2 &= Pobj(Xobj,Yobj,Zobj) \\ P &= Pshad(Xshad,Yshad,Zshad) \end{aligned}$$

Since the shadow is on the ground, Yshad = 0 and the expression:

$$\begin{aligned} (y-y1)/(y2-y1) &= (Yshad-Y1sc)/(Yobj-Y1sc) = k2 \\ x &= k1 * (x2-x1) + x1 = Xshad = k1 * (Xobj-X1sc) + X1sc \\ z &= 0 = Zshad \\ x &= k1 * (x2-x1) + x1 = Xshad = X2 * (Zobj-Z1sc) + Z1sc \end{aligned}$$

The line is from the light through an object point. The point where this line intersects the ground becomes a point of the shadow,  $P_{\text{shad}}(X_{\text{shad}}, 0, Z_{\text{shad}})$ . Note that the shadow is a object, a set of points.

Shadows cast on one object by another are likely to be ugly, garble the form, and are rarely used by fine artists, so no cast shadow algorithm has been implemented for this article. You can, however, create cast shadows by first creating the shadow of an object on the floor and then projecting this shadow object, considered as a floor "pattern," onto the far side of the to-be-shadowed object.

The algorithm for rendering floor, or ground, points also uses the two-point form of the equations of a line in space. However, because the ground, or floor, can have infinitely many points the algorithm works backward from those for other objects. Instead of picking a point on the floor and checking to see if it is visible the algorithm picks a point on the frame and checks to see if it should reflect a floor point. This avoids having to calculate non visible points. See Illustration 1.2(a). The algorithm constructs a line through a point on the frame and the eye, and sees where it would intersect the floor. The points are

```
P1 = Pframe(Xframe, Yframe, Zframe) = Pframe(col, row, 0)
P2 = Peye(Xeye, Yeye, Zeye)
P = P11x(Xf1r, Yf1r, Zf1r) = P11x(Xf1r, 0, Zf1r)
```

For a given row and column on the frame we wish to calculate whether it should reflect a floor point.

Since  $Yf1r = 0$ :

$$(y-y1)/(y2-y1) = -row/(Yeye-row) = k \text{ Also since } Zframe = 0 \\ (z-z1)/(z2-z1) = Zf1r/Zeye = k \\ Zf1r = k*Zeye \text{ and } Zf1r = x1*(Key-col) + col$$

The point  $(Xf1r, 0, Zf1r)$  can be used to determine what color  $P_{\text{frame}}(\text{col}, \text{row}, 0)$  should be. The determination could involve examining the floor pattern and the effect of light at that floor point.

## The Code

Most functions have a separate program to make it easier to customize and re-compile the code. Some objects, however, have too many points to write out. For example, a floor. The floor program given below generates a floor point, places it, rotates it, colors it, lights, it and renders it all without the point ever leaving temporary storage. Cubes or boxes also have large numbers of points. To avoid repeatedly handling the object's files, the 'cuber' program generates, colors, positions and lights the cube, point by point. The cuber program could be extended to generate the cube's shadow, and render both cube and shadow. You can use the programs as separate functions, writing out the sets of points between each operation, and using scripts to orchestrate production, or you can string the code of different functions together and compile special purpose programs for each class of object you use.

To get help on running a program, type its name at the CLI prompt. The program will write instructions to the screen on how it is to be run. Some programs use scripts either to copy all necessary files to ram; or to process a numbered sequence of object's points files. Scripts are give in Table 1.2. Some programs require a parameter file. Example 'params' files are given in Table 1.1

Following are a brief descriptions of the programs. Further comments are in the listings.

Zbuffer, Listing 1, reads files of object's points and plots appropriate points on the Amiga screen. Zbuffer gets the eye coordinates and the maximum z distance from an ASCII parameter file, which you can prepare and/or edit using 'ed.' Zbuffer's parameter file is called 'zbparams'. Note that you can keep a copy of the parameter's definitions and input format at the end of the 'params' file.

Zbuffer produces two types of files, one with a 'rend' suffix and others with a 'zbuf' suffix. The first is the rendered screen image, the others are lists of z values for non-zero screen points. The 'zbuf' file has the same format as an object's point file but the contents are completely different. The 'zbuf' file record refers to the Amiga screen. It consists of four 16-bit integers: the pixel color, screen column, screen row, and associated z value.

Zbuffer accepts an indeterminate number of miscellaneous named object files. This facility together with 'compzbufs' program, which merges two or more 'zbuf' files, allows the user to build up a virtually unlimited number of objects in limited storage.

For processing speed zbuffer script copies all the files zbuffer will need to RAM: (where you have placed the files of object's points).

Use the programs as they are  
or customize the algorithms  
and compile  
custom variations.

Zbuffer requires a list of input files on the command line. Note that you can continue the list past the end of the line if you do not hit return. To run Zbuffer on three files, for example, type at the CLI prompt:

```
zbuffer <filename> <filename2> <filename7> 9999
```

The file '9999' is an 'end-of-files' file. When zbuffer tries to process 9999, it recognizes that it is finished. The 9999 file should contain one set of dummy values.

Zbuffer, as do all of the programs described in this article, writes a record of its activity and run parameters to a 'Run\_Notes' file on disk.

Compzbufs, Listing 2, reads two or more 'zbuf' files and combines these into one new 'rend' file and new, composite, 'zbuf' files. To run compzbufs, at the CLI prompt, type:

```
compzbufs <'zbuf' 111> <'zbuf' 111>2> ... 9999
```

You can also use compzbufs to new 'zbuf' files.

Compzbufs script copies necessary files to ram: Compzbufs does not require a parameter file.

Shadowgen, Listing 3, reads an object's points and given a light

position, calculates the points of the plane  $v=0$  that fall in shadow. It writes these points to files of a new shadow object. Note that the shadow is an object. Since the ground is frequently patterned the program also calculates the shadow color according to the floor pattern.

Shadowgen requires a pattern repeat image. How the repeat is used is described in the discussion of 'floor', Listing 4, the program that generates a patterned ground.

Shadowgen requires a parameter file called 'shparams' containing the light position, and the width and height of the pattern repeat. The parameter file also contains 16 numbers which specify color register re-mapping, if any. The numbers zero to 15 will leave the color registers unchanged. It, for example, position three in the list, which corresponds to color register two, contains the digit '9' then the program will color any point having color identity '3' with the color in register '9'. This is handy for changing the colors of the repeat image.

Zfloor, Listing 4, generates and renders a patterned ground. The familiar checkered ground of cyberspace has become a bit boring. That is unfortunate because patterned grounds provide powerful 3-D visual

rendering. Alternatively, the rendered ground image could be 'pasted' behind the rendered objects using the backer program, Listing 7.

Besides requiring the pattern image, zfloor program requires a parameter file, 'flparams,' containing light position, eye position, pattern repeat width and height, and 'yop,' the maximum z distance.

Illustration 1.3 shows the rendered ground for the repeat shown in Illustration 1.2. The ground in the illustration is lighted, but since lighting requires a long explanation, the code for lighting the floor is not included in Listing 4. It will be given in Part II of this article. Meanwhile you can use the program to try out different patterned grounds.

Cuber, Listing 5, generates rectangular boxes. It reads height, width, and depth, row and column increments, color and offsets, from a parameter file and generates points of a hollow box. Since boxes of any size at all have large numbers of points cuber, as well as generating points, places and lights (the lighting code will be given in Part II) the box to avoid handling files of points. You may want to expand the code to generate the box's patterned shadow, and then render both box and shadow. A point can then go from generation through rendering without ever leaving temporary storage. That is how I generated the boxes of Illustration 1.3, which shows three boxes floating in space with shadows on a patterned ground.

To run cuber, type at the CLI prompt:

```
cuber <output box-object's name>
```

For many boxes their points as projected unto the frame are not dense enough to produce a solid looking image. You can get around this by generating points for a box, say, twice as large as you want, then just before lighting the points, scale the box's dimensions down to half. This will increase the density of points.

Placer, Listing 6, positions objects within a scene. It reads a point, scales, rotates, translates, then writes the revised point to a new file.

Explanations of the transformations are not given here. My article, "Make Your Own 3-D Vegetation Objects," in Volume 3, Number 1 of this journal, gives detailed explanations of the code for scaling, translation, and rotation.

Placer rotations do not include correction for non-square pixel distortion because different resolutions, different monitors, and different printers, will require different correction amounts. Correction code is described in my article, "Transformer" in Volume 3, Number 1 of this journal. To get the inches/pixel of your particular resolution-monitor-printer setup draw an image of a one pixel frame in a point program, print it, and measure the resulting width and height. Divide these by the corresponding pixel counts.

To run placer, copy the files of object's points to ram., then use the 'placer.script.' Type at the CLI prompt:

```
Execute placer.script <objectname>
```

Backer, Listing 7, reads a backdrop image and pastes it behind rendered objects. The backdrop fills only pixels of the foreground having color register zero. The backdrop image could be a rendered ground or even a partially rendered scene. Or, the backdrop image could be a flat backdrop picture. A common old Fine Arts technique was to paste a distance scene behind a 3-D foreground, with no middle ground. The backdrop blue for Illustration 1.3, 'Floating Cubes,' was pasted behind the rendered patterned floor. Purists who object to

**The code  
is designed to work  
on parts of the scene inde-  
pendently.**

clues. So this floor program allows you to create elaborate floor patterns instead of the simple checkers. Elaborate patterns can have surprising rendered images so a little experimentation will repay your effort. For example, one pattern I found looks exactly like patio carpeting, another looks like breeze ripples on a lake, another suggests cyberspace flying-saucer hangers!

Zfloor gets the pattern color of a floor point by the point's position on the pattern repeat, modula the repeat's width and depth. See Illustration 1.2(b). You can make the image of a pattern repeat in a paint program. The program looks for the repeat in the upper left corner of the image.

As the program creates a virtual patterned ground, it produces the zbuffer rendering of how the floor would look from the given point of view in all directions and to the horizon. This is a special way of dealing with a huge object, in this case the ground far and wide, without ever writing its points to files. Since a ground will even so have a large number (perhaps 30 or more) of 'zbuf' files the program does not produce them automatically. You must key 'z' during the run to get 'zbuf' files. Since, however, you can use the 'rend' file directly in your final image you will seldom need the 'zbuf' files for a floor.

The rendered floor's 'zbuf' files can be composited with 'zbuf' files of objects using compzbufs as described above, to make up the final



superimposition of rendered parts of an image can, of course, compose the entire image from 'zbuf' files. Superposition, however, saves a great amount of work and even makes feasible effects that would otherwise require prohibitive amounts of storage. A combination of means works best. Where image elements intermingle their 'zbuf' files can be combined using compzbufs and then non-intermingled elements added behind by pasting.

Addpoints, Listing 8, reads object's points, plots them on the screen, then accepts user mouse-clicks indicating points to be added to the object. Although it would be possible to build an entire object using addpoints, the program is meant to be used for editing objects. For example, a few points might need to be added to fill a hole. Or, a few points added with addpoints can indicate the beginning and end of a line whose points can then be written into the ASCII points file using 'ed.' Addpoints can also be helpful in adding sub-objects when building complex objects. It will record your click of the center point coordinates of the to-be-added sub-objects. The added points file, called '<inputfilename>.ap' is written to ram. To run addpoints, type at the CLI prompt:

```
addpoints <object-filename>
```

Addpoints requires a parameter file called 'adparams' containing the color register to be assigned the added points. Addpoints has an x-y screen and a z-y screen which give a side and front view for positioning points. You will need a top view, of the x-z plane, to position points on a shadow object, which lies in the y = 0 plane. Rather than elaborate the algorithm you can make, by a few small changes, a special version of Addpoints, which has a x-z plane and an x-y plane.

## Tutorial

Follow these activities to get started running these programs. First to make a cube, use 'ed' to make up a parameter file called 'cuparams' or copy the 'cuparams' file from Table 1.1. Try the values:

```
100 3 0.6 0.4 0.0 260 100 860
3 6 9 100 100 560 1.0 1.0 1.0
```

Reading from left to right, top to bottom, the cube is to have side length of 100 pixels. The increment of two means only every other line will be included, it will not be opaque. It will be rotated 0.6 radians about the x-axis and 0.4 radians about the y-axis. Its center is (260,100,860). The top and bottom will have the color in register 3, front and back will have the color in register 6, and left and right side will have color in register 9. The light is at (100,800,560). (Although the cube program doesn't have its lighter algorithm so it won't light the cube keep the paramere formate intact and all ready to add the lighter code fragment to be given in Part II) All the sides will be the same length as given in the first parameter. These definitions and input format are always available in the program listing and you can keep a copy on the parameter file itself below the values. Put the 'cuparams' file in the same directory as cuber. At the CLI prompt type:

```
cuber cube
```

When 'cuber' is finished it will have written to ram: four files called cube1, cube2, cube3, and cube4.

Leave the four files in ram. Make a 'zbparams' file with the parameter values 240 200 430 1800 which will give an eye position =

(240,200,430) and will cut the ground off at z = 1800.

Next type at the CLI prompt:

```
Execute zbuffer.script zbuffer
```

The script will copy everything you need for a zbuffer run to ram: and will change directory to ram: It will report "Ready to run zbuffer" At the CLI prompt type:

```
zbuffer mycube cube1 cube2 cube3 cube4 9999
```

where 'mycube' is the name of the output rendering and zbuffer files, and '9999' is an end-of-files file.

When zbuffer is finished you will find a file called 'mycuberend' in ram. There will also be one or more files 'mycubezbuf1', 'mycubezbuf2', etc.

To save the rendered image covert it to ILBM format and write it to disk. At the CLI prompt type:

```
ram2ilbm mycuberend dfl:mycuberend.pic hi 4
```

**The rendering algorithm  
is a Z-buffer algorithm,  
which is much faster than  
ray tracing.**

Copy the 'zbuf' files to another disk so you can combine them with other 'zbuf' files later.

Next, make a shadow for the cube. Shadowgen will make a patterned shadow in case the shadow falls on a patterned floor. Although we will feed shadower a pattern image we will trick the program into giving a shadow of color register 15. Make up a 'shparams' file with the following parameter values.


```
100 800 560
0 15 15 15 15 15 15 15 15 15 15 15 15 15 15 15
168 168
```

The parameters show the light at the same place as for the cube above. Light at (100,800,560). The string of 16 numbers map every color on the input pattern except black into color register 15. The width and height of the pattern repeat will be 168 since we will use the pattern image on disk. It has width and height of 168 pixels.

At the CLI prompt type:

```
Execute shadow.script shadowgen
```

This will copy everything you need to ram: and change directory to ram: When it is finished copying it will print:



Who  
can you turn to  
for the  
best coverage  
of the fast-paced  
Amiga  
market?

# Amazing Computing of course!

**Amazing Computing** for the Commodore Amiga, **AC's GUIDE** and **AC's TECH** provide you with the most comprehensive coverage of the Amiga. Coverage you would expect from the longest running monthly Amiga publication.

100 pages of **Amazing Computing** bring you insights into the world of the Commodore Amiga. You'll find comprehensive reviews of Amiga products, complete coverage of all the major Amiga trade shows, and hints, tips, and tutorials on a variety of Amiga subjects such as desktop publishing, video programming, & hardware. You'll also find a listing of the latest Fred Fish disks, monthly columns on using the CLI and working with AREXX, and you can keep up to date with new releases in New Products and other neat stuff.

**AC's GUIDE** to the Commodore Amiga is an indispensable catalog of all the hardware, software, public domain collection, services and information available for the Amiga. This amazing book lists over 3500 products and is updated every six months!

**AC's TECH** for the Commodore Amiga provides the Amiga user with valuable insights into the inner workings of the Amiga. In-depth articles on programming and hardware enhancement are designed to help the user gain the knowledge he needs to get the most out of his machine.



## Call 1-800-345-3360

"Ready to run shadowgen"

At the CLI prompt type:

```
shadowgen cubeshad pattern.pic cube1 cube2 cube3 cube4 9999
```

Do not hit return until after you type '9999'

This will produce several files called 'cubeshad1', 'cubeshad2' ... etc. in ram. Run zbuffer on them as above. Use the same 'zparams' file.

And so on. You will be able to continue on your own. To find out what is needed on the command line, type the name of the program at the CLI prompt.

Try using 'compzbufs' to combine the shadow and cube 'zbuf' files. Save the rendered cube + shadow image. Make a plain floor of color in register 13. Use backer to put the floor image behind the cube + shadow image. Try out 'zflood' on some of your own patterns which you can make in a paint program. Be sure to keep a note of the width and depth of you pattern repeats.

### Biographical Sketch

Laura M. Morrison has a master's degree in mathematics from New York University, NY. She has worked as an Operations Research Analyst designing applications software for Esso R&E, Union Carbide and Eastern Airlines.

Ms Morrison studied painting at the Art Student's League in New York and the Academie Julian in Paris.

## Listing 1

```
/* zbuffer.c Listing 1
Reads lighted and positioned object's points,
a color map and a point of view.
Renders the object accordingly.
Produces 'zbuf' and 'rend' files.
Copyright 1993 by Laura M. Morrison */
#include "stdio.h"
#include "libraries/dosextens.h"
#include "intuition/intuition.h"
#include "intuition/intuitionbase.h"
#include "exec/exec.h"
#include "math.h"
#define COLORS12 32
#define DEPTH 4L
#define WIDTH 640L
#define HEIGHT 400L
#define MAXPOINTS 4000L
extern struct Window *w;
extern struct RastPort *rp;
extern struct Screen *OpenScreen();
struct FileHandle *Open();
USHORT cmap[16];
SHORT *cols;
struct NewScreen newscreen = {
    0,0,640,400,4,0,1,
```

```
HIRESIMAGE,
CUSTOMSCREEN|CUSTOMBITMAP,
NULL,NULL,
NULL,NULL,
};
struct Screen *scr;
struct NewWindow nw = {
    420,30,210,50,0,1,
    CLOSEWINDOW|VANILLAKEY,
    WINDOWCLOSE|ACTIVATE|BORDERLESS|WINDOWDRAG,
    NULL, NULL,
    "idcmp",
    NULL, NULL,
    0,0,100,50,
    CUSTOMSCREEN
};
struct Window *w;
struct BitMap *bitm;
struct RastPort *rp;
struct ViewPort *vp;
struct IntuitionBase *IntuitionBase;
struct GfxBase *GfxBase;
static WORD *zptr[400];
WORD *rowptr;
WORD *buf;
static float kl, xo, yo, zo, xe, ye, ze;
static float xs, ys, zs, xw, yw, zw;
static float xsf, ysf, zsf, inside;
char outfile[120];
char infile[120];
char str[120];
main(argc, argv)
int argc;
char *argv[];
{
    LONG ofh, afh, bfh, cfh, pth;
    struct IntuiMessage *mess;
    ULONG class;
    USHORT code;
    int row, col, cc, i, j, k;
    int count, count2, dif, number, yon;
    float xscal, yscal, zscal;
    int xeye, yeye, zeye, xobj, yobj, zobj;
    int xscr, yscr, zscr;
    int xoff, yoff, zoff, xwld, ywld, zwld;
    int dm = 999;
    WORD zval, oldval;
    LONG planesize, butsize, nb = 0;
    LONG nt, colorsize;
    FILE *fopen(), *wp, *zp, *lp, *pp;
    colorsize = COLORS12;
    count2 = 0;
    IntuitionBase = (struct IntuitionBase *)
        OpenLibrary("intuition.library",0);
    if (IntuitionBase == NULL)
    {
        printf("Couldn't open Intuition library\n");
        exit(1);
    }
    GfxBase = (struct GfxBase *)
        OpenLibrary("graphics.library",0);
    if (GfxBase == NULL)
    {
        printf("Couldn't open Graphics library\n");
        exit(2);
    }
    if (argc < 3)
```

```

    printf(" Need name of output file and \n");
    printf(" one or more object's points files, \n");
    printf(" and an 'eof' file called '9999'\n");
    exit(3);
}
if ((pp) = fopen("zbparams", "r")) == NULL)
{
    printf(" Need a 'zbparams' file with eye position.\n");
    exit(4);
}
fscanf(pp, "%d %d %d %d %f %f %f %f", &xeye, &yeye, &zeye, &yon);
fclose(pp);
printf("\n\n\n\n\n\n\n\n\n\n");
printf(" ");
printf("RENDER an OBJECT/OBJECTS\n");
printf(" ");
printf(" from a Point of View \n\n");
printf(" ");
printf("Copyright 1993 by Laura M. Morrison\n");
printf("\n\n\n\n\n\n\n\n\n\n");
/* set up storage for the z value of each point
on the screen. */
for(i = 0; i < HEIGHT; i++)
{
    bufsize = 2*WIDTH;
    zptr[i] = (WORD *)
    AllocMem(bufsize, MEMF_FAST|MEMF_CLEAR);
    if(zptr[i] == NULL)
    {
        printf("No memory for zbuffer row %d.\n", i);
        goto quit;
    }
}
/* set all the z values to the maximum */
for(i=0; i< HEIGHT; i++)
{
    rowptr = zptr[i];
    for (j=0; j < WIDTH; j++)
    {
        *(rowptr++) = yon;
    }
}
/* Set up a CustomBitMap and open a screen
to receive the rendered image. */

bitm=(struct BitMap *)AllocMem
(sizeof(struct BitMap), MEMF_CHIP);
if (bitm==NULL)
{
    printf("Couldn't allocate bmap struct \n");
    goto quit;
}
InitBitMap(bitm, DEPTH, WIDTH, HEIGHT);
planesize=(LONG) (WIDTH/8)*HEIGHT;
for (i=0; i < DEPTH; i++)
{
    bitm->Planes[i]=(PLANEPTR)
        AllocRaster(WIDTH, HEIGHT);
    if (bitm->Planes[i] == NULL)
    {
        printf("bitmap planes were null\n");
        goto quit;
    }
    BitClear (bitm->Planes[i].planesize, 1);
}
newscr.CustomBitMap = bitm;

```

```

scr=(struct Screen *)OpenScreen(&newscr);
if (scr==NULL)
{
    printf("scr was null\n");
    goto quit;
}
rp = &scr->RastPort;
vp = &scr->ViewPort;
nw.Screen = scr;
w = OpenWindow(&nw);
if (w==NULL) goto quit;
/* Read in a special palette file called 'pal'. */
pfh = Open("pal", MODE_OLDFILE);
if (pfh != NULL)
{
    cols = cmap[0];
    nb = Read(pfh, cols, colorsize);
    Close(pfh);
    LoadRGB4(vp, cmap, 16);
}
SetRast(rp, 0);
number=1;
nextfile;;
number++;
printf(" ");
printf("Now processing file %s\n", argv[number]);
dif = strcmp(argv[number], "9999");
if (dif==0)
{
    goto finish;
}
strcpy(infile, argv[number]);
zp = fopen(infile, "r");
if (zp == NULL)
{
    goto nextfile;
}
readmore;;
fscanf(zp, "%d %d %d %d\n",
        &cc, &xobj, &yobj, &zobj);
if (cc > 997)
{
    fclose(zp);
    goto nextfile;
}
count++;
xe = (float)xeye;
ye = (float)yeye;
ze = (float)zeye;
zo = (float)zobj;
xo = (float)xobj;
yo = (float)yobj;
if ( (zo-ze) != 0 )
{
    k1 = -ze/(zo-ze);
    xsf = k1*(xo-xe) + xe;
    ysf = k1*(yo-ye) + ye;
    zsf = 0.0;
    xwld = WIDTH - (int) xsf;
    ywld = HEIGHT - (int) ysf;
    xscr = xwld;
    yscr = HEIGHT - ywld;
    if ((xscr > 0) && (xscr < WIDTH)
        && (yscr > 0) && (yscr < WIDTH))
    {
        rowptr = zptr[yscr];
        oldval = *(rowptr + xscr);
    }
}

```

```

    if (zobj < oldval)
    {
        rowptr = zptr[yscr];
        *rowptr = xscr + zobj;
        SetAPen(rp,cc);
        WritePixel(rp,xscr,yscr);
    }
}
/* end of eye inside object */
while((mess = struct InquiMessage *)
    GetMsg(w->UserPort)) != NULL)
{
    class = mess->Class;
    code = mess->Code;
    ReplyMsg(mess);
    if (class == CLOSEWINDOW) goto finish;
    if ((class == VANILLAKEY) && (code == 'q'))
        goto quit;
}
goto readmore;
finish:
if (w)
{
    CloseWindow(w);
    w = NULL;
}
ScreenToBack(scr);
printf("\n\n\n      Finished ... \n\n");
printf("\n\n");
printf("Writing output files to ram: ... \n\n");
strcpy(outfile,"ram:");
strcat(outfile,argv[1]);
strcat(outfile,".rend");
ofh = Open(outfile,MODE_NEWFILE);
if (ofh != NULL)
{
    for (i=0; i < DEPTH; i++)
    {
        nb = Write(ofh,bitm->Planes[i],planesize);
        cols = Acmap[0];
        nb = Write(ofh,cols,colsize);
        Close(ofh);
    }
    number = 1;
    count2 = 0;
    itoa(number,stri);
    strcpy(outfile,"ram:");
    strcat(outfile,argv[1]);
    strcat(outfile,".zbuf");
    strcat(outfile,stri);
    wp = fopen(outfile,"w");
    /* Save z values as this may not be the final
    rendering of the entire picture. Z values may
    be needed to make a composite of two partial
    renderings. */
    for(row = 0; row < HEIGHT; row++)
    {
        for(col = 0; col < WIDTH; col++)
        {
            rowptr = zptr[row];
            cc = ReadPixel(rp,col,row);
            zval = *(rowptr + col);
            if (cc != 0)
            {
                fprintf(wp," %d %d %d %d\n",cc,col,row,zval);
            }
        }
    }
}

```

```

    count2++;
    /* When output file gets large close it and
    open another. */
    if (count2 > MAXPOINTS)
    {
        fprintf(wp," %d %d %d %d\n",
            dm,dm,dm,dm);
        fclose(wp);
        count2=0;
        number++;
        itoa(number,stri);
        strcpy(outfile,"ram:");
        strcat(outfile,argv[1]);
        strcat(outfile,".zbuf");
        strcat(outfile,stri);
        wp = fopen(outfile,"w");
    }
}
fprintf(wp," %d %d %d %d\n",dm,dm,dm,dm);
fclose(wp);
quit:
if (w) CloseWindow(w);
if (scr)
{
    CloseScreen(scr);

    for (i = 0; i < DEPTH; i++)
    {
        if (bitm->Planes[i] > 0)
        {
            FreeRaster(bitm->Planes[i],WIDTH,HEIGHT);
        }
    }
    if (bitm) FreeMem(bitm,sizeof(struct BitMap));
}
bufsize = 2*WIDTH;
for(i = 0; i < HEIGHT; i++)
{
    if(zptr[i])
    {
        FreeMem(zptr[i],bufsize);
    }
}
lp = fopen("dfl:Run_Notes","a");
if (lp != NULL)
{
    fprintf(lp,"\n\n %s %s \n", argv[0],argv[1]);
    fprintf(lp," %d %d %d %d\n",xcye,yeye,zeye,yonl);
    fclose(lp);
}
if (IntuitionBase) CloseLibrary(IntuitionBase);
if (GfxBase) CloseLibrary(GfxBase);
/* end of main */
/* itoa is from Kernighan and Ritchie
C Programming */
itoa(nc,stri);
char str[];
int nt;
{
    int c, j, i, sign;
    if ((sign = nc) < 0)
        nt = -nt;
    i=0;
    do {
        str[i++] = nt % 10 + '0';
    } while (nt /= 10);
    if (sign < 0)
        str[i++] = '-';
    str[i] = '\0';
}

```

```

    } while (cnt - 1 < 0);
    if (sign < 0)
        str[i++] = '-';
    str[i] = '\0';
    for (i = 0; j = strlen(str) - 1; i++, j--)
    {
        c = str[j];
        str[j] = str[i];
        str[i] = c;
    }

    return(0);
}

```

## Listing 2

\* mpzhuf.c Listing 2  
 Combines 'zbuf' files.  
 Copyright 1993 by Laura M. Morrison.

```

#include "stdio.h"
#include "libraries/dosextens.h"
#include "intuition/intuition.h"
#include "intuition/intuitionbase.h"
#include "exec/exec.h"
#include "math.h"
#define COLORSIZ 32
#define BUFSIZE 32000L
#define DEPTH 41
#define WIDTH 6401
#define HEIGHT 400L
#define MAXPOINTS 4000L
extern struct RastPort *rp;
extern struct Screen *OpenScreen();
struct FileHandle *Open();
USHORT cmap[16];
USHORT *cols;
static struct NewsScreen newscl = {
    0, 0, 640, 400, 4, 0, 1,
    HIRFS_LACE,
    CUSTOMSCREEN|CUSTOMBITMAP,
    NULL, NULL,
    NULL, NULL,
};
struct Screen *scr;
struct BitMap *bitm;
struct RastPort *rp;
struct ViewPort *vp;
struct IntuitionBase *IntuitionBase;
struct GfxBase *GfxBase;
static WORD xval, yval;
static WORD *aptr[400];
WORD *rowptr;
WORD *buf;
static int yon, zobj;
char del[120];
char outfile[120];
char infile[120];
char str[120];
main(argc, argv)
int argc;
char *argv[];

LONG oth, pth;

```

```

int count, row, col, cc, i, j, k;
int dummy, dif, number, count2;
int xobj, yobj, zobj, xscr, yscr, zscr;
int dm = 999;
LONG planesize, bufsiz, nb, nc;
LONG nt, colorsiz, success;
FILE *fopen(), *zp, *ap, *lp, *pp, *wp;
colorsiz = COLORSIZ;
yon = 20000;
if (argc < 3)
{
    printf("Need name of output file, then '\n");
    printf("names of one or more 'zbuf' files.\n");
    printf("then an 'eof' file called '9999'\n");
    exit(101);
}
IntuitionBase = (struct IntuitionBase *)
    OpenLibrary("intuition.library", 0);
if (IntuitionBase == NULL)
{
    printf("Couldn't open Intuition library\n");
    exit(1);
}
GfxBase = (struct GfxBase *)
    OpenLibrary("graphics.library", 0);
if (GfxBase == NULL)
{
    printf("Couldn't open Graphics library\n");
    exit(2);
}
printf("\n\n\n\n\n\n\n\n");
printf(" ");
printf("MERGE ZBUFFER FILES\n");
printf(" ");
printf("Copyright 1993 by Laura M. Morrison\n");
printf("\n\n\n\n\n\n\n\n");
bitm = (struct BitMap *)
    AllocMem(sizeof(struct BitMap), MEMF_CHIP);

if (bitm == NULL)
{
    printf("Couldn't allocate bmap struct\n");
    goto finish;
}
InitBitMap(bitm, DEPTH, WIDTH, HEIGHT);
planesize = (LONG) (WIDTH/8) * HEIGHT;
for (i = 0; i < DEPTH; i++)
{
    bitm->Planes[i] = (PLANEPTR)
        AllocRaster(WIDTH, HEIGHT);
    if (bitm->Planes[i] == NULL)
    {
        printf("bitmap planes were null\n");
        goto finish;
    }
    BltClear(bitm->Planes[i], planesize, 1);
}
newscl.CustomBitMap = bitm;
scr = (struct Screen *) OpenScreen(&newscl);
if (scr == NULL)
{
    printf("scr was null\n");
    goto finish;
}
rp = &scr->RastPort;
vp = &scr->ViewPort;
pfh = Open("pal", MODE_OLDFILE);

```

```

if (pfh != NULL)
{
    cols = &cmmap[0];
    nb = Read(pfh,cols,colorsiz);
    Close(pfh);
    LoadRGB4(vp,cmmap,16);
}
SetRGB4(vp,0,0,0,0);
SetRast(rp,0);
/* Set up a buffer to save a z value
for each screen pixel. */
for (i = 0; i < HEIGHT; i++)
{
    bufsize = 2*WIDTH;
    zptr[i] = (WORD *)AllocMem(bufsize,
                                MEMF_FAST|MEMF_CLEAR);
    if (zptr[i]==NULL)
    {
        printf(" No memory for row %d\n",i);
        goto quit;
    }
}
for (i = 0; i < HEIGHT; i++)
{
    rowptr = zptr[i];
    for (j = 0; j < WIDTH; j++)
    {
        *(rowptr++) = yon;
    }
}
number = 1;
nextzbuf;
number++;
printf(" ");
printf("Processing file %s\n",argv[number]);
dif = strcmp(argv[number],"9999");
if (dif==0)
{
    goto finish;
}
strcpy(infile,argv[number]);
ap = fopen(infile,"r");
if (ap==NULL)
{
    printf(" Couldn't find 'zbuf' %s ... \n");
    goto nextzbuf;
}
readmorea;
fscanf(ap,"%d %d %d %d\n",
        &cc,&col,&row,&zobj);
if (cc > 997)
{
    fclose(ap);
    ap = NULL;
    goto nextzbuf;
}
rowptr = zptr[row];
oldval = *(rowptr + col);
if (zobj < oldval)
{
    rowptr=zptr[row];
    *(rowptr + col) = zobj;
    SetAPen(rp,cc);
    WritePixel(rp,col,row);
}
goto readmorea;
finish;

```

```

ScreenToBack(scr);
printf("\n\n");
printf("Finished ... Writing output files.\n");
printf("\n\n\n\n\n");
strcpy(outfile,"ram:");
strcat(outfile,argv[1]);
strcat(outfile,".rend");
ofh = Open(outfile,MODE_NEWFILE);
if (ofh != NULL)
{
    for (i=0; i < DEPTH; i++)
    {
        nb = Write(ofh,bitm->Planes[i],planesize);
    }
    cols = &cmmap[0];
    nb = Write(ofh,cols,colorsiz);
    Close(ofh);
}
number = 1;
count2 = 0;
itoa(number, str);
strcpy(outfile,"ram:");
strcat(outfile,argv[1]);
strcat(outfile,"zbuf");
strcat(outfile,str);
wp = fopen(outfile,"w");
printf(" ");
printf("Writing output file %s\n",outfile);
for (row = 0; row < HEIGHT; row++)
{
    for (col = 0; col < WIDTH; col++)
    {
        rowptr = zptr[row];
        cc = ReadPixel(rp,col,row);
        zval = *(rowptr + col);
        if (cc != 0)
        {
            fprintf(wp,"%d %d %d %d\n",cc,col,row,zval);
            count2++;
        }
        /* When output file gets large close it
        and open another. */
        if (count2 > MAXPOINTS)
        {
            fprintf(wp,"%d %d %d %d\n",
                    dm,dm,dm,dm);
            fclose(wp);
            wp = NULL;
            count2=0;
            number++;
            itoa(number, str);
            strcpy(outfile,"ram:");
            strcat(outfile,argv[1]);
            strcat(outfile,"zbuf");
            strcat(outfile,str);
            wp = fopen(outfile,"w");
            if (wp==NULL)
            {
                goto quit;
            }
            else
            {
                printf(" ");
                printf("Writing output file %s\n",
                        outfile);
            }
        }
        /* start a new file */
    }
}

```



```

        * : not zero */
    }
    fprintf(wp, "%d %d %d %d %d", dm, da, da, da, dm);
    fclose(wp);
    wp = NULL;
quit:
    if (!ap) fclose(ap);
    if (!wp) fclose(wp);
    if (scr)
    {
        CloseScreen(scr);
        for (i = 0; i < DEPTH; i++)
        {
            if (bitm->Planes[i] > 0)
            {
                FreeRaster(bitm->Planes[i], WIDTH, HEIGHT);
            }
        }
        if (!bitm) FreeMem(bitm, sizeof(struct BitMap));
    }
    bufsize = 2*WIDTH;
    for (i = 0; i < HEIGHT; i++)
    {
        if (zptr[i])
        {
            FreeMem(zptr[i], bufsize);
        }
    }
    if (!fp) fclose(fp);
    if (fp != NULL)
    {
        fprintf(fp, "n");
        for (i = 0; i < number; i++)
        {
            fprintf(fp, " %s ", argv[i]);
        }
        fclose(fp);
    }
    if (IntuitionBase) CloseLibrary(IntuitionBase);
    if (GfxBase) CloseLibrary(GfxBase);
    /* end of main */
    *toas is from Keogh and Richie */
    toa(nt, str);
    clear(str);
    nt = 0;

    int c, j, i, sign;
    if ((sign = nt) < 0)
        nt = -nt;
    i = 0;
    do {
        str[i++] = nt % 10 + '0';
    } while ((nt /= 10) > 0);
    if (sign < 0)
        str[i++] = '-';
    str[i] = '\0';
    for (i = 0, j = strlen(str) - 1; i < j; i++, j--)
    {
        c = str[i];
        str[i] = str[j];
        str[j] = c;
    }
    return(0);
}

```

## Listing 3

```

/* shadowgen.c Listing 3.
Generates points for the shadow
on the floor of an input object.
Copyright 1993 by Laura M. Morrison */
#include "stdio.h"
#include "libraries/dosextens.h"
#include "intuition/intuition.h"
#include "intuition/intuitionbase.h"
#include "exec/exec.h"
#include "math.h"
#define COLORSIZ 32L
#define DEPTH 3L
#define WIDTH 640L
#define HEIGHT 400L
#define MAXPOINTS 4000L
/* map1[] and map2[] assign color registers to points
given a color id number from one to forty-five. More
explanation in Part II of this article. */
static int map1[] = {0, 1, 2, 3, 4, 5, 6, 7,
                    8, 9, 10, 11, 12, 13, 14, 15,
                    1, 2, 3, 1, 2, 3, 1, 2, 3,
                    4, 5, 6, 4, 5, 6, 4, 5, 6, 7, 8, 9,
                    1, 8, 9, 10, 11, 12, 13};
static int map2[] = {0, 1, 2, 3, 4, 5, 6, 7,
                    8, 9, 10, 11, 12, 13, 14, 15,
                    4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15,
                    7, 8, 9, 10, 11, 12, 13, 14, 15, 10, 11, 12,
                    13, 14, 15, 13, 14, 15, 1};
static int colid[16];
USHORT cmap[16];
extern struct Screen *scr;
struct NewScreen newscl = {
    0, 0, 640, 400, 4, 0, 3,
    HiresILACE,
    CUSTOMSCREEN|CUSTOMBITMAP,
    NULL, "Converting libm image to raw format",
    SWLL, NULL,
    1};
struct Screen *scr;
SHORT *cols;
extern struct RastPort *rp;
struct FileHandle *Open();
struct BitMap *bitm;
struct RastPort *rp;
struct ViewPort *vp;
struct IntuitionBase *IntuitionBase;
struct GfxBase *GfxBase;
static int wx, wz, ecc, cput;
static int xlit, ylit, zlit;
static int row, col, i, j, k;
static int i1, count2, number, numbering;
static int dm, dummy;
static int zobx, xohj, yobj, zobj;
static int zshad, xshad, yshad, zshad;
static char toraw[120];
static char intile[120];
static char outtile[120];
static char deltile[120];

```





```

c = str[i];
str[i] = str[i+1];
str[i+1] = c;
}
return(0);

```

## Listing 4

### \* zfloor.c Listing 4

Reads floor pattern, generates points for full floor, lights points, renders floor. Note: This version does not include the lighting code which will be given in Part II of this article.

Copyright 1993 by Laura M. Morrisson \*

```

#include "stdio.h"
#include "libraries/dosextens.h"
#include "intuition/intuition.h"
#include "intuition/intuitionbase.h"
#include "exec/exec.h"
#include "math.h"
#define COLORSIZ 321
#define DEPTH 4L
#define WIDTH 640L
#define HEIGHT 400L
#define MAXPOINTS 4000L
/* Zone[] will be needed for the lighting code
to be added in Part II. */
static float zone[] = { 0.0,
0.02618, 0.05236, 0.07854, 0.10472, 0.13090,
0.15708, 0.18326, 0.20944, 0.23562, 0.26180,
0.28798, 0.31416, 0.34034, 0.36652, 0.39270,

0.41888, 0.44506, 0.47124, 0.49742, 0.52360,
0.54978, 0.57596, 0.60214, 0.62832, 0.65450,
0.68068, 0.70686, 0.73304, 0.75922, 0.78540,

0.81158, 0.83776, 0.86394, 0.89012, 0.91630,
0.94248, 0.96866, 0.99484, 1.02102, 1.04720,
1.07338, 1.09956, 1.12574, 1.15192, 1.17810,
1.20428, 1.23046, 1.25664, 1.28282, 1.30900,

1.33518, 1.36136, 1.38754, 1.41372, 1.43990,
1.46608, 1.49226, 1.51844, 1.54462, 1.57080 };
/* 'maps' let you change color register numbers */
static int map1[] = { 0, 1, 2, 3, 4, 5, 6, 7,
8, 9, 10, 11, 12, 13, 14, 15,
1, 2, 3, 1, 2, 3, 1, 2, 3,
4, 5, 6, 4, 5, 6, 4, 5, 6,
7, 8, 9, 7, 8, 9, 10, 11, 12, 13 };
static int map2[] = { 0, 1, 2, 3, 4, 5, 6, 7,
8, 9, 10, 11, 12, 13, 14, 15,
4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15,
7, 8, 9, 10, 11, 12, 13, 14, 15,
10, 11, 12, 13, 14, 15, 13, 14, 15, 13 };
static int colid[18];
USHORT cmap[16];
extern struct Window *w;
extern struct Screen *scr;
extern struct Screen *scr2;
/* Set up one input and one output screen. */
struct NewScreen newscreen = {
0, 0, 640, 400, 4, 0, 1,

```

```

HIDES_FACE,
CUSTOMSCREEN + CUSTOMHIDE,
NULL, /* Converting lib image to raw ... */
NULL, NULL,
};
struct Screen *scr1;
struct NewWindow nw = {
0, 0, 30, 10, 50, 0, 1,
CLOSEWINDOW + VISIBLEKEY,
WINDOWBASE | ACTIVE | BORDERLESS + WINDOWBAG,
NULL, NULL,
"cdmp",
HIDES_FACE,
0, 0, 100, 50,
CUSTOMSCREEN
};
struct Window *w;
struct NewScreen newscreen2 = {
0, 0, 640, 400, 4, 0, 1,
HIDES_FACE,
CUSTOMSCREEN + CUSTOMHIDE,
NULL, NULL,
NULL, NULL,
};
struct Screen *scr2;
struct RastPort *rp;
extern struct RastPort *rp2;
struct FileHandle *Open();
struct BitMap *bmap;
struct BitMap *bmap2;
struct RastPort *rp;
struct RastPort *rp2;
struct ViewPort *vp;
struct ViewPort *vp2;
struct IntuitionBase *IntuitionBase;
struct GfxBase *GfxBase;
static WORD *zptr[400];
static WORD *zmap;
static WORD *zbuf;
static float k1, k3, x1rf, y1rf, z1rf;
static float xeyf, yeyf, zeyf, rowt, colt;
static int x1ir, y1ir, z1ir, wx, wz;
static int co, end, cpat, cob;
static int xobj, yobj, zobj;

```



The complete set of listings and source code, along with the illustrations and tables mentioned in the article can be found on the AC TECH disk.

Please write to  
**Laura Morrisson**  
**c/o AC's TECH**  
**P.O. Box 2140**  
**Fall River, MA 02722**

# StateMachine

by J.T. Steichen

StateMachine is intended to allow an Amiga programmer to create finite state machines with a graphical user interface, rather than generate the program by hand. StateMachine does not generate the action functions associated with the finite state machine. This is up to the user of StateMachine.

A state machine is a program that executes a given function (the action) when a given input is supplied (the state), then it moves to another state (the next state) and waits for more input. This input is supplied by a user-supplied function that essentially is a lexical analyzer-generator (scanner). A 'scanner' takes an input file and breaks it up into lexemes (i.e., keywords, whitespace, punctuation, etc). Based on the type of lexeme, it returns a value to the finite state machine, which in this case, is called the State. The state machine generated uses re-directed I/O for input; therefore, usage of your state machine program will be:

```
your statemachine <input_definition_file> <output_source_file>
```

**NOTE:** you decide if the output is redirected by how you define your output functions!

In a way, your computer itself is a large state machine, with instructions serving as lexemes, execution of the instructions serving as actions, and the operating system determining what the next state of the PC is!

So when exactly would a programmer use a finite state machine instead of something straightforward, like a large case statement or a series of if-then-else constructs? For those of you who have examined a simple *Microsoft Windows 3.x* program, you will notice that all data communications between the operating system and a windows application is done via messages. Hundreds of them! Every *WinMain()* function has to have a large case statement (or call a function that does) in order to cover every message that the program will respond to. Even for simple programs, this case statement can be quite large!

*Intuition* also uses messages, sent to the *JDCM* port attached to all open windows.

Fortunately, Intuition has fewer messages to contend with, so it's much easier to use case statements for decoding what the user is doing with the GUI. Unfortunately, the IDCMP message values are not in a linear sequence! This rules out using finite state machines for the GUI decoding (IDCMP message values are powers of two. This means a finite state machine might have a very large ( $2^{32}$ ) next-state/action matrix with very few used elements. This is known as a sparse matrix).

If IDCMP message values were linear, this is how a IDCMP message decoder could be constructed using a finite state machine:

FROM TYPICAL IDCMP MESSAGE LOOP,

```
while (looping == TRUE)
{
    if (!message = (struct IntuiMessage *)
        GetMsg(my window->DevicePort)) == NULL)
    {
        (void) Wait(1L + my window->ep_dlight);
        continue;
    }
    mclass = message->Class;
    mcode = message->Code;
    (void) ReplyMsg(message);
    switch (mclass)
    {
        case MOUSEBUTTONS:
            DoMouseButtons(mcode);
            break;
        case KEMURPICK:
            DecodeMenuSelection(mcode);
            break;
        case GADGETUP:
            DecodeGadgetSelection(mcode);
            break;
        case MENUSIZE:
            UpdateGraphics();
            break;
    }
}
```

```
case KEMURPICK:
    UpdateGraphics();
    break;
case CLOSERBUTTON:
    /* Clear menu strip if there is one */
    DoClearMenu();
    looping = FALSE;
default:
    break;
}
```

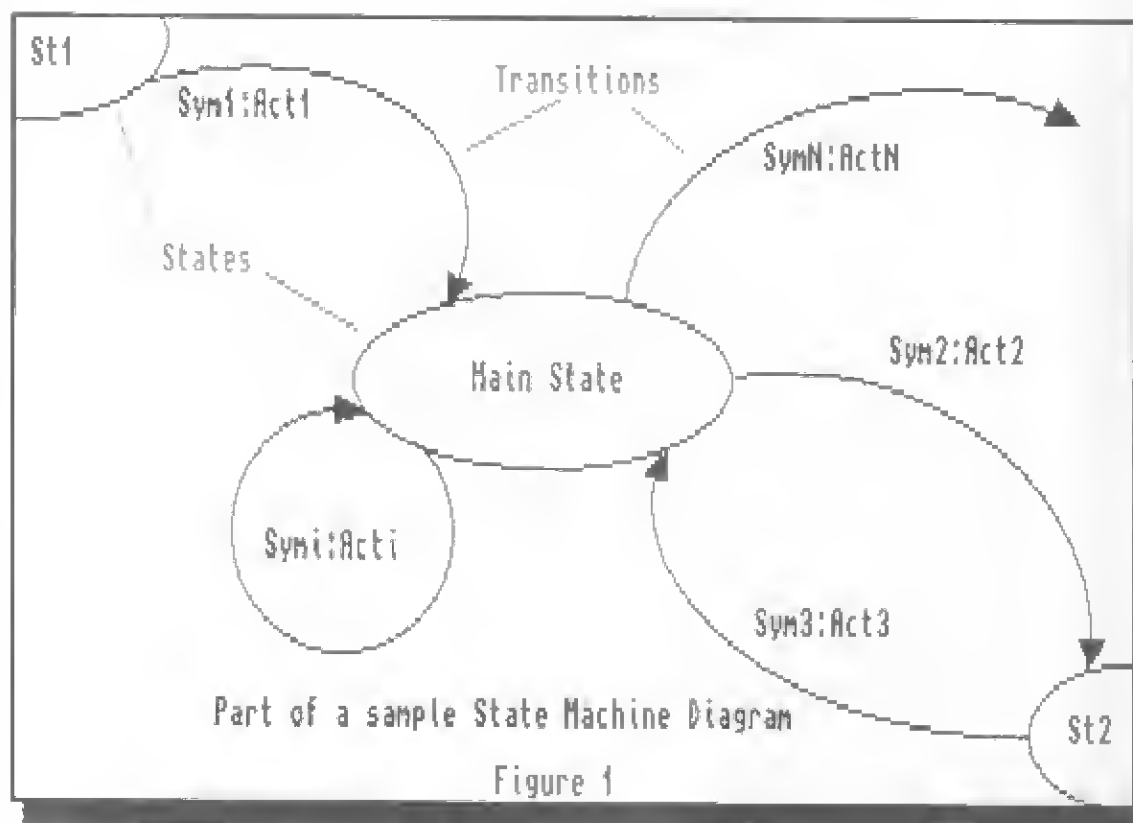
CHANGED INTO A FINITE STATE MACHINE:

```
while (looping == TRUE)
{
    if (!message = (struct IntuiMessage *)
        GetMsg(my window->DevicePort)) == NULL)
    {
        (void) Wait(1L + my window->ep_dlight);
        continue;
    }
    state = message->Class;
    mcode = message->Code;
    (void) ReplyMsg(message);
    /* IDCMP FunctionTable[mcode][state].action();
    looping = (*IDCMP.FunctionTable[mcode][state].nextstate);
    */
}
```

As you can see, the code is much simpler, but much harder to trace and debug!

The places a programmer might find uses for finite state machines are file translators, interpreters, or compilers. This means that you could create a programming language of your own, for example, a simple interpreted language that uses Amiga graphics functions to draw shapes, etc. This is precisely what I have chosen for my example—take a look at Example.tsm, which reads simple graphic instructions from an input file and executes them. Example recognizes the following instructions:

EXAMPLE



# Table 1

Done, Cleanup, Begin, OpenGraph, Error, RptErr,	Begin, Ignore, DefMove, Ignore, Begin, Ignore,	Begin, Ignore, DefDraw, Ignore, Begin, Ignore,	Begin, OpenGraph, DefColor, Ignore
Error, RptErr, Error, RptErr, DefColor, StowNum,	DefColor, Ignore, Error, RptErr, DefColor, Ignore,	Begin, ChgColor, Error, RptErr, DefColor, Ignore,	Error, RptErr, Error, RptErr,
Error, RptErr, Error, RptErr, DefMove, StowNum,	DefMove, ignore, Error, RptErr, DefMove, Ignore,	Begin, MoveFunc, Error, RptErr, Begin, Ignore,	Error, RptErr, Error, RptErr,
Error, RptErr, Error, RptErr, DefDraw, StowNum,	DefDraw, Ignore, Error, RptErr, DefDraw, Ignore,	Begin, DrawFunc, Error, RptErr, Begin, Ignore,	Error, RptErr, Error, RptErr,
Done, Cleanup, Begin, OpenGraph, Error, RptErr,	Begin, Ignore, DefMove, Ignore, Begin, Ignore,	Begin, Ignore, DefDraw, Ignore, Begin, Ignore,	Begin, OpenGraph, DefColor, Ignore,
Done, Cleanup, Error, RptErr, Done, Cleanup,	Done, Cleanup, Error, RptErr, Done, Cleanup,	Done, Cleanup, Error, RptErr, Done, Cleanup	Error, RptErr, Error, RptErr,

```
LORES;
COLOR Red8,Red8,Green8,Blue8;
MOVE X8,Y8;
DRAW X8,Y8;
```

The first thing a programmer has to figure out is what lexemes the scanner must recognize and return to the finite state machine. For the example, the scanner must recognize:

numbers, commas, semicolons, whitespace, newlines, 'MISEE', 'LORES', 'COLOR', 'MOVE', 'DRAW' and EOF.

Next, the programmer must decide what to do when the scanner returns a given lexeme. These are the action functions that will be placed into the action, next-state table. Some actions are always going to be needed, these are:

```
error(), ignore(), echo();
```

The following actions are added to Example:

StowNum, DrawFunc, MoveFunc, OpenGraph, ChgColor and Cleanup as well as RptErr and Ignore.

Start up the StateMachine program and wait for the Transition Editor to complete initializing. Select Add from the Actions menu and enter the following actions into the requester by typing the string into the string gadget and pressing the Add button gadget:

Ignore, RptErr, StowNum, DrawFunc, MoveFunc, OpenGraph, ChgColor, and Cleanup.

After all strings show up in the list gadget, click on the 'Done Adding' button. Once you are back in the Transition Editor, select Add from the Symbols menu. Enter the following strings into the string gadget in the same manner as you used for the Actions:

EndFile, WhiteSpace, Semicolon, Rline, Lores, Move, Draw, Color, Number, Comma, Endline.

**NOTE:** These are entered in this order since the generated finite state machine expects to see the values as the scanner defines them. Only the Symbols and NextState,Actions entries are order-dependent.

Return to the Transition Editor and select Add from the States menu. Enter the following state names into the requester in the same manner as used on the Actions and Symbols:

Begin, DefColor, DefMove, DefDraw, Error, and Done.

Return to the Transition Editor and get ready to do the major task of entering in the NextState,Actions

name pairs. This can be done by first double-clicking on the State name and then double-clicking on the Action name that you want to appear in the list. StateMachine doesn't care how many newlines are in the NextState/Actions list, so space them so that you can read them (See Table 1)

After all NextState/Actions are entered into the list gadget, we are ready to enter the prologue and code sections of the Example state machine. Press the 'Goto Prolog Editor' Gadget and wait for the prolog editor. Enter the following.

```
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <intuition/base.h>
void Ignore(); /* Your compiler will complain if
```

these are not here! \*/

```
void RptErr();
void ShowMsg();
void DrawFunc();
void MoveFunc();
void OpenGraph();
void ChgColor();
void Closeup();
```

NOTE: All code is already typed in Example.fsm and Example.c

Finally, we are ready for the Code Editor. Press the 'Goto Code Editor' button. Enter the following C source code into the list gadget:

```
extern char *idstr; /* defined in the scanner */
int linecount = 0; /* used for error reporting */
void Ignore()
{
    if (idstr[0] == '\n')
        linecount++;
    return;
}
```

```

}

void RptErr()
{
    fprintf(stderr, "ERROR in input stream\n");
    fprintf(stderr, "Found on line %d\n", linecount);
    return;
}

struct NewScreen ns = {
    0, 0, 640, 200, 4, 0xFF, 0xFF, 0xFF, CUSTOMSCREEN,
    NULL, (UBYTE *) "StateMachine Example", NULL, NULL
};

struct NewWindow nw = {
    0, 0, 640, 200, 0xFF, 0xFF, CLOSEWINDOW,
    WINDOWCLOSE | ACTIVATE | SMART_REFRESH,
    NULL, NULL, (UBYTE *) "StateMachine Example:",
    NULL, NULL, 20, 20, 640, 200, CUSTOMSCREEN
};

extern struct IntuitionBase *IntuitionBase;
struct GfxBase *GfxBase = NULL;
struct Window *example_window = NULL;
struct Screen *example_screen = NULL;
void OpenGraph()
{
    if (strcmp(idstr, "LORES") == 0)
    {
        ns.ViewModes = 0;
        ns.Depth = 3;
        ns.Width = 320;
        ns.Height = 200;
        nw.Width = 320; /* Make window lores sized! */
        nw.Height = 200;
        nw.MaxWidth = 320;
        nw.MaxHeight = 200;
    }
    if (!IntuitionBase) {struct IntuitionBase *;
        OpenLibrary("Intuition.library", 0L) == NULL}
        return;
    if (!GfxBase) {struct GfxBase *;
        OpenLibrary("graphics.library", 0L) == NULL}
        {
            CloseLibrary(IntuitionBase);
            return;
        }
    if (example_screen = (struct Screen *) OpenDevice() andn0) == NULL)
    {
        CloseLibrary(GfxBase);
        CloseLibrary(IntuitionBase);
    }
}
```

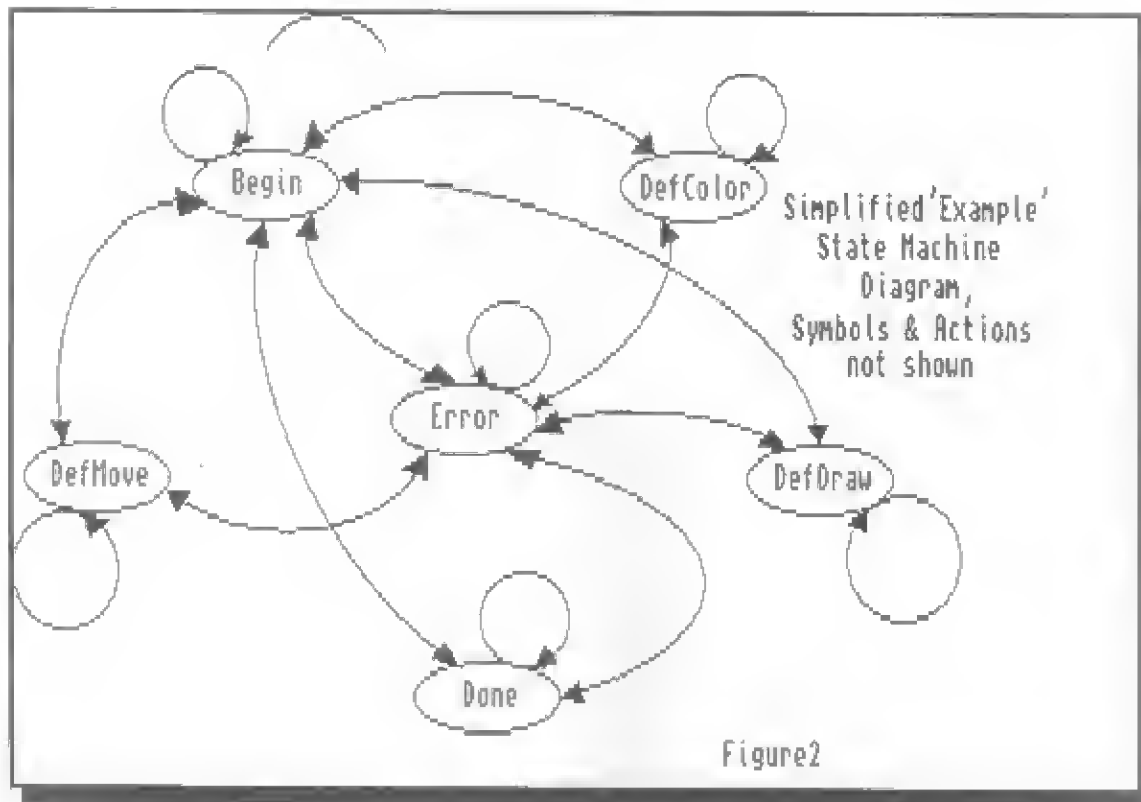


Figure2



```

    return;
}
aw.Screen = example_screen;
if (!example_window = (struct Window *) OpenWindow( andw )) == NULL)
{
    CloseScreen( example_screen );
    CloseLibrary( GDBase );
    CloseLibrary( IntuitionBase );
    return;
}
return;
}

int  num[5] = { -1, -1, -1, -1, 0 };
void DrawFunc() /* store numbers for DrawFunc, MoveFunc and
ChgColor */
{
    extern int atoi( char * );
    int index = 0;
    while (true[index] >= 0)
        index++;
    num[index] = atoi( idatr );
    return;
}

void DrawFunc()
{
    if (example_window != NULL)
        Draw( example_window->RPort, num[0], num[1],
            num[2], num[3] );
    num[0] = num[1] = -1;
    return;
}

void MoveFunc()
{
    if (example_window != NULL)
        Move( example_window->RPort, num[0], num[1] );
    num[0] = num[1] = -1;
    return;
}

void ChgColor()
{
    extern struct ViewPort *ViewPortAddress( struct Window * );
    if (example_window != NULL)
    {
        SetRGB( ViewPortAddress( example_window ), num[0], num[1],
            num[2], num[3] );
        SetAPen( example_window->RPort, num[0] );
    }
    num[0] = num[1] = num[2] = num[3] = -1;
    return;
}

#define TRUE
#define TRUE 1
#define FALSE 0
#define FALSE 0

void Cleanup()
{
    int Checked = TRUE;
    class = 0;
    struct IntuiMessage *message = NULL;
    char att[256], *title = endatt[0];

    if (example_window == NULL)
        goto SkipLoop;
    strcpy( title, aw.Title );
    strcpy( title, " Click on close window gadget to EXIT!" );
    SetWindowTitles( example_window, title, NULL );

    while (Checking == TRUE)
    {
        if (message = (struct IntuiMessage *)
            GetMsg( example_window->UserPort )) != NULL)
        {
            (void) Wait( 1L << example_window->UserPort->exp_sigBit );
            continue;
        }
        class = message->Class;
        (void) ReplyMsg( message );
        switch( class )
        {
            case CLOSEWINDOW:
                Checking = FALSE;
                break;
            default:
                break;
        }
    }
}

SkipLoop:
    if (example_window != NULL)
        CloseWindow( example_window
}

```

```

if (example_screen != NULL)    CloseScreen( example_screen );
if (GDBase != NULL)          CloseLibrary( GDBase );
if (IntuitionBase != NULL)    CloseLibrary( IntuitionBase );
return;

```

Now, return to the Transition Editor by pressing the 'Done Editing' gadget. Select Save As from the Project menu and save your new state machine source code. Finally, select 'Generate FSM' from the Project menu. This will turn your definition file into C-source code that can be compiled into an executable program, once it is linked with the scanner. That's all there is to producing finite state machines!

## NOTES:

The files included with StateMachine are as follows:

- FSM\_Aux - Do NOT change the name of this file!
- StateMachine.h - Title screen, Do NOT change the name of this file!
- StateMachine - The executable program, Do NOT change the name of this program!
- StateMachine.info - Do NOT change the name of this file!
- C FSMGen - The Finite State Machine generator program, Do NOT

change the name of this program! It can be placed in any directory you wish as long as the ToolTypes in the icon are changed to reflect its location.

Change CmdDirectory=C: to wherever you locate FSMGen.

- FSMTest - A test input file.
- FSMTest.IdealOutput - what FSMGen should produce from FSMTest!
- FSM\_Manual - Description of gadgets and menus in StateMachine.
- Article - Description of what StateMachine does, along with a tutorial.
- MakeFile - SMake file for the Lattice C compiler to work on Example.
- Example.fsm - a tutorial state machine definition file.
- Example.c - the output of FSMGen when Example.fsm is supplied.
- Example.flex - a tutorial state machine scanner definition file.
- Example.lex.c - the output of FLeX when Example.flex is supplied.
- TestEx.input - input file for checking and debugging Example program.

It is interesting to note that FSMGen is itself a finite state machine. It was generated from a more primitive version of itself!

The StateMachine GUI was created with *CanDo* V2.0. I recommend *CanDo* highly since it cuts down on how much C code I have to write in order to make an effective Amiga-based program.

StateMachine was inspired by a couple of articles in the *C User's Journal* entitled:

'Building a Finite State Machine' by R. Ward, *C User's Journal*, February 1989, pgs. 94-99.

'A Finite State Machine Generator' by R. Ward, *C User's Journal*, April 1989, pgs. 101-107.

→

# SUMMER 1993 AC'S GUIDE

Looking for a specific product for your Amiga but you don't know who makes it? Want a complete listing of all the Fred Fish software available?

Just looking for a handy reference guide that's packed with all the best Amiga software and hardware on the market today?

If so, you need *AC's GUIDE for the Commodore Amiga*. Each *GUIDE* is filled with the latest up-to-date information on products and services available for the Amiga. It lists public domain software, user's groups, vendors, and dealers. You won't find anything like it on the planet; and you can get it only from the people who bring you the best monthly resource for the Amiga, *Amazing Computing*.

So to get all this wonderful information, call 1-800-345-3360 today and talk to a friendly Customer Service Representative about getting your *GUIDE*. Or stop by your local dealer and demand your copy of the latest *AC's GUIDE for the Commodore Amiga*.

## List of Advertisers

Company	Page Number
Delphi Noetic	18
Devine Computers	13
Digital Creations	CIV
Dineen Edwards	CII

## WHAT'S ON IT?

### AC's TECH 3.3 Disk Includes Source & Executables For:

- REXX Rainbow Library Programs
- Programming the Amiga in Assembly
- All You Ever Wanted to Know About Morphing
- StateMachine
- Programming in PostScript
- Bill Nee's Correction
- AC's GUIDE  
Product Registration Form

## AND MORE!

# AC's TECH 3.3 CHECK IT OUT!

# AC's TECH Disk

## Volume 3, Number 3

### *A few notes before you dive into the disk!*

- You need a working knowledge of the AmigaDOS CLI as most of the files on the AC's TECH disk are only accessible from the CLI.
- In order to fit as much information as possible on the AC's TECH Disk, we archived many of the files, using the freely redistributable archive utility 'lharc' which is provided in the C: directory. lharc archive files have the filename extension .lzh.

To unarchive a file *foo.lzh*, type *lharc x foo*

For help with lharc, type *lharc ?*

*Also, files with 'lock' icons can be unarchived from the WorkBench by double-clicking the icon, and supplying a path.*



We pride ourselves in the quality of our print and magnetic media publications. However, in the highly unlikely event of a faulty or damaged disk, please return the disk to **PM Publications, Inc.** for a free replacement. Please return the disk to:

AC's TECH  
Disk Replacement  
P.O. Box 2140  
Fort River, MA 02720-2140

**Be Sure to  
Make a  
Backup!**

### **CAUTION!**

Due to the technical and experimental nature of some of the programs on the AC's TECH Disk, we advise the reader to use caution, especially when using experimental programs that initiate low level disk access. The entire liability of the quality and performance of the software on the AC's TECH Disk is assumed by the purchaser. PM Publications, Inc., their distributors, or their retailers, will not be liable for any direct, indirect, or consequential damages resulting from the use or misuse of the software on the AC's TECH Disk. (This agreement may not apply in all geographical areas.)

Although many of the individual files and directories on the AC's TECH Disk are freely redistributable, the AC's TECH Disk itself and the collection of individual files and directories on the AC's TECH Disk are copyright PM Publications, Inc., and may not be duplicated in any way. The purchaser, however, is encouraged to make an archival backup copy of the AC's TECH Disk.

Also, be extremely careful when working with hardware projects. Check your work twice to avoid any damage that can happen. Also, be aware that using these projects may void the warranties of your computer equipment. PM Publications, or any of its agents, is not responsible for any damages incurred while attempting this project.

Unfortunately, the code in the articles didn't work, and since it was intended to be portable, it certainly didn't make use of the Amiga's graphical user interface capabilities. My rewrite of the code from these articles corrects both of these deficiencies. FSMGen, which is based on these articles, was created from a crude working version of the articles' code and a more complex lexical analyzer (the scanner). FSMGen uses a unique syntax of its own for the input definition file, which actually generates the C source code, which is subsequently compiled by you, the user. You don't have to remember what this syntax is, since StateMachine generates the definition file for you. The syntax is similar to what LeX and YACC use for their definition files.

LeX - Lexical analyzer-generator program, a UNIX programming tool. There is a public-domain program called Flex, available on Fred Fish disk #407 that works real well (be sure that the source code skeleton files don't get mixed up with the skeleton files that have to be included in your lexical analyzer, the file names are similar!). YACC - Yet Another Compiler Compiler, a UNIX programming tool. There is a public-domain program called Bison, available on a Fred Fish disk also, that does the same thing as YACC.

## example.c

```

/* Start of Prologue */

/* FSMGen Source file written by StateMachine */
/* Copyrighted 1992, by J. T. Steichen */

.....
** NAME: Example<input_file>
**
** WARNINGS: Since all scanners generated by FLeX used
re-directed input.
**          This Example program can only be run from
the CLI properly!
**          (Unless you want to type the input_file by
hand!)
**
** See the MakeFile for info on how to turn Example.fsm
into Example.
**
** See the Article for the syntax that Example expects to
see in the
** input file.
.....

#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include <unistd.h>
#include <math.h>

void Ignore();
void RptErr();
void ShowNum();
void DrawFunc();
void MoveFunc();

```

```

void OpenGraph();
void ChgColor();
void Cleanup();

/* Bottom of Prologue */

/* Start of Transition table */

struct Transitions {
    int      nextstate;
    void      (*act)();
};

/* Defines for Symbol names: */

#define EMPTY -1
#define EndFile 0
#define WhiteSpace 1
#define SemiColon 2
#define HiRes 3
#define LoRes 4
#define Move_ 5
#define Draw_ 6
#define Color 7
#define Number 8
#define Comma 9
#define EndLine 10

/* Defines for State names: */

#define Begin 0
#define DefColor1
#define DefMove 2
#define DefDraw 3
#define Error 4
#define Done 5
#define STOP 5

struct Transitions fsm_table[11] = {

/*
    EndFile      WhiteSpace      SemiColon
HiRes
    LoRes        Move_          Draw_
Color
    Number       Comma          EndLine
*/

/* Begin */

Done ,Cleanup, Begin ,Ignore , Begin ,Ignore , Begin
,OpenGraph,
Begin ,OpenGraph, DefMove,Ignore , DefDraw,Ignore ,
DefColor,Ignore ,
Error ,RptErr , Begin ,Ignore , Begin ,Ignore ,

/* DefColor */

Error ,RptErr , DefColor,Ignore , Begin ,ChgColor, Error
,RptErr ,
Error ,RptErr , Error ,RptErr , Error ,RptErr , Error
,RptErr ,
DefColor,ShowNum, DefColor,Ignore , DefColor,Ignore ,

/* DefMove */

Error ,RptErr , DefMove,Ignore , Begin ,MoveFunc, Error

```

```

,RptErr ,
Error ,RptErr , Error ,RptErr , Error ,RptErr , Error ,RptErr ,
DefMove,ShowMap, DefMove, Ignore, Begin ,Ignore ,

* DefDraw *

Error ,RptErr , DefDraw, Ignore , Begin , DrawFunc, Error ,RptErr ,
Error ,RptErr , Error ,RptErr , Error ,RptErr , Error ,RptErr ,
DefDraw,ShowMap, DefDraw, Ignore, Begin , Ignore ,

* Error *

Done ,Cleanup, Begin ,Ignore , Begin ,Ignore , Begin ,OpenGraph,
Begin ,OpenGraph, DefMove, Ignore , DefDraw, Ignore , DefColor, Ignore ,
Error ,RptErr , Begin , Ignore , Begin , Ignore ,

* Done *

Done ,Cleanup, Done ,Cleanup, Done ,Cleanup, Error ,RptErr ,
Error ,RptErr , Error ,RptErr , Error ,RptErr , Error ,RptErr ,
Done ,Cleanup, Done ,Cleanup, Done ,Cleanup

:

typedef int      FSMTYPE;

static int      mstate, token;

/* Code for FSM Start_Function(): */

FSMTYPE      main()
{
    mstate = Begin;
    token = EMPTY;

    do
    {
        token = yylex();
        if ( fsm_table[ mstate ][ token ], !act );
        mstate = fsm_table[ mstate ][ token ];
    } while (mstate != STOP);

    return( 0 );
}

/* Start of Code Section: */

extern char * istr;

int linecount = 0;

void Ignore()
{
    if ( istr[0] != '\n' )
        linecount++;
    return;
}

void RptErr()

```

```

{
    fprintf( stderr, "Error in Input: %s\n", istr );
    fprintf( stderr, "Found on Line %d\n", linecount );
    return;
}

struct NewScreen ns =
{
    0, 0, 640, 200, 0, 0xFF, 0xFF, 0xFF, CUSTOMSCREEN,
    NULL, 1024*1024, "IntuitionMachine Example", NULL, NULL
};

struct NewWindow nw =
{
    0, 0, 640, 200, 0xFF, 0xFF, CLOSEWINDOW,
    WINDOWCLOSE + ACTIVATE + SMART_REFRESH,
    NULL, NULL, 1024*1024, "IntuitionMachine Example",
    NULL, NULL, 20, 20, 640, 200, CUSTOMSCREEN
};

extern struct IntuitionBase *IntuitionBase;
struct GfxBase *GfxBase = NULL;

struct Window *example_window = NULL;
struct Screen *example_screen = NULL;

void OpenGraph()
{
    if ( !strcmp( istr, "looked" ) )
    {
        ns.ViewMode = 0;
        ns.Depth = 5;
        ns.Width = 320;
        ns.Height = 200;
        nw.Width = 320; /* Make window as small as possible */
        nw.Height = 200;
        nw.MaxWidth = 320;
        nw.MaxHeight = 200;
    }

    if ( !IntuitionBase = (struct IntuitionBase *)
        OpenLibrary( "intuition.library", 0 ) )
        return;

    if ( !GfxBase = (struct GfxBase *)
        OpenLibrary( "graphics.library", 0 ) )
        return;

    CloseLibrary( IntuitionBase );
    return;
}

if ( !example_screen = (struct Screen *) OpenScreen( &ns ) )
    return;

    CloseLibrary( GfxBase );
    CloseLibrary( IntuitionBase );
    return;
}

nw.Screen = example_screen;
if ( !example_window = (struct Window *) OpenWindow( &nw ) )
    return;

    CloseScreen( example_screen );
    CloseLibrary( GfxBase );
    CloseLibrary( IntuitionBase );
    return;
}

```



```

#define SEMIolon 1
#define HIRES 3
#define LORES 4
#define MOVE 5
#define DRAW 6
#define COLOR 7
#define NUMBER 8
#define COMMA 9
#define EOL 10

char NIL[256], *idstr 64000;
int i = 0;

#ifdef EXAMPLE_DEBUG

int main(void)
{
    int tokval = 0;

    tokval = yylex();
    while (tokval >= 0)
    {
        switch( tokval )
        {
            case EOL:      fprintf( stderr, "\n" );
                           break;
            case WHITE:    fprintf( stderr, "ts", idstr );
                           break;
            case HIRES:    fprintf( stderr,
                                "%s", idstr );
                           break;
            case LORES:    fprintf( stderr,
                                "%s", idstr );
                           break;
            case EOF_:     fprintf( stderr, "End of
File\n" );
                           return 0;
            case SEMICOLON: fprintf( stderr,
                                ":" );
                           break;
            case MOVE:     fprintf( stderr,
                                "%s", idstr );
                           break;
            case NUMBER:   fprintf( stderr, "%s", idstr );
                           break;
            case DRAW:     fprintf( stderr,
                                "%s", idstr );
                           break;
            case COLOR:    fprintf( stderr,
                                "%s", idstr );
                           break;
            case COMMA:    fprintf( stderr,
                                ":", idstr );
                           break;
            default:       fprintf( stderr, "intokval =
%d\n", tokval );
                           fprintf( stderr, "idstr =
%s\n", idstr );
                           break;
        }
        tokval = yylex();
    }
    return 0;
}

```

```

#endif /* EXAMPLE_DEBUG */

}

%%

[\n]      { idstr[0] = '\n';
            idstr[1] = '\0';
            return EOL;
}

[ ]+      { i = 0;
            while (*yytext + i) == ' ')
            {
                idstr[i] = ' ';
                i++;
            }
            idstr[i] = '\0';
            return WHITE;
}

[\t]      { i = 0;
            while (*yytext + i) == '\t')
            {
                idstr[i] = ' ';
                i++;
            }
            idstr[i] = '\0';
            return WHITE;
}

[\\:]     { idstr[0] = ':';
            idstr[1] = '\0';
            return SEMICOLON;
}

"HIRES"   { strcpy( idstr, "HIRES" );
            return HIRES;
}

"LORES"   { strcpy( idstr, "LORES" );
            return LORES;
}

"MOVE"    { strcpy( idstr, "MOVE" );
            return MOVE;
}

"DRAW"    { strcpy( idstr, "DRAW" );
            return DRAW;
}

"COLOR"   { strcpy( idstr, "COLOR" );
            return COLOR;
}

","       { strcpy( idstr, "," );
            return COMMA;
}

[0-9]+    { sscanf( yytext, "%s", idstr );
            return NUMBER;
}

"-"[0-9]+ { sscanf( yytext, "%s", idstr );
            return NUMBER;
}

"+"[0-9]+ { sscanf( yytext, "%s", idstr );
            return NUMBER;
}

[0x1A]    { return EOF_; }

%%

```

(continued on page 62)

# Programming in PostScript

## PostScript Programming

PostScript is a programming language that is of major importance to virtually every piece of productivity software in existence, yet few programmers know anything about it. To be sure it is not the new challenger to 'C', BASIC or ARexx. Yet PostScript is running on computers around the world that incorporate the very latest of technologies and fastest processors. PostScript came in the back door, and will always stay there since it is the language of printers and graphics. But every major professional-level productivity application has to use PostScript to generate high quality output. The time has come for you to take the plunge. Join me then in this three-part voyage into the world of PostScript.

### What is it?

What is PostScript? "The PostScript language is a simple interpretive programming language with powerful graphics capabilities. Its primary application is to describe the appearance of text, graphical shapes, and sampled images on printed or displayed pages." This is the concise definition given in *The PostScript Language Reference Manual* from Adobe Systems Incorporated. So what does that mean in real life? PostScript is a full programming language that is specially designed for describing how a printed page or display will be rendered. When I say full programming language I mean that all the normal operators for math, string manipulation, I/O, and even stack manipulation are present.

But what does describing a page mean? To most programmers, writing the command `lineto(10,20)` means draw a line from the current point to the point at 10,20. The command executes the same way in nearly all circumstances and is essentially the same as the rendering. In PostScript there are similar commands, but the environment is different. When a `lineto` command is given, the only assumption is that relative to the rest of the image a line of some sort will be rendered to some point that is relatively known as 10,20. With PostScript it is assumed that the drawing commands that are originally given will be modified, possibly at several levels before being rendered to some device. That device may be a screen, slide maker, imagesetter, or laser printer. The device may be able to render color, grayscale, or simply black and white. The idea is that the description (the drawing commands) are independent of the file output device.

In fact a `lineto` does not even cause a line to be rendered. Instead it adds a segment to an overall path. A path is a list of the drawing commands that were used to create it. A path then

can be used to draw an outline, fill a shape, clip to a shape, or even be used to place shapes along it. In PostScript, the creation of an object and its rendering are well removed from each other.

Another facet of PostScript that is important to mention is that it is an interpreted language. From a practical standpoint this means that PostScript can be very slow. It also means that PostScript does not really run programs in the traditional sense. Instead the PostScript interpreter accepts commands and executes them. When there are no more commands it waits around. Getting to the end of a list of commands does not cause anything to happen, like a page printing for instance. This works out well because this allows one piece of PostScript code to be included inside another without any changes. As we will see this is a very important feature.

### Under the Hood

Now that we know a little about the ideas behind PostScript, let's look at it from a programmer's standpoint. PostScript is a stack-oriented interpreted language not unlike FORTH. This means that as items are sent to the interpreter, they are placed on a stack until they are used. The stacks are Last in First Out (LIFO). When a command is encountered it takes items off the stack and places the results (if any) back on the top of the stack. For instance the 'C' command `x = 4 + 5`, would result in the PostScript code `/x 5 4 add def`. The `/x` places the variable `x`'s name on the stack. The 5 and 4 wait on the stack until the `add` operator replaces them with 9. Then the `def` operator accepts a value then a name (remember, last in first out) and binds the name to that value. The `def` command does not generate a result so the stack is now back to the state it was before the example.



# A look inside the PostScript Language

by Dan Weiss

This stack works much the same as the stack used by high level languages for passing parameters. The differences are that you have no access to the stack pointer, and virtually everything is handled via the stack as opposed to variables. To help out with using the stack, PostScript includes the following stack manipulation commands: pop, exch, dup, copy, index, roll, clear, count, mark, clearmark, countmark. These operators all work on the operand stack. The operand stack is the one used in the example, and the one used by most PostScript operators. There are, however, other stacks—the dictionary stack, execution stack, and graphics state stack.

## Hello, World

Now that we understand the basics of how PostScript works, let's write the PostScript equivalent of the infamous "Hello, World" program. In PostScript we draw a box.

```
newpath
100 100 moveto
200 100 lineto
200 200 lineto
100 200 lineto
100 100 lineto
stroke
showpage
```

If you have access to a PostScript printer, or a clone like POST, then stop and enter the program. If you did it right you should have generated a square about 1.5" inches from the bottom left corner, and it should be about 1.5" square. For those without printers or POST, the box looks like this. Congratulations, you just wrote your first piece of PostScript software. Let's take a quick look at it.

The first command newpath always needs to be called when there is no existing path. This initializes the current path to be empty. Now that path has been initialized, we have to give a starting point so we send down 100 100 moveto. The lineto commands work pretty much as expected, but remember, they do not cause a line to be drawn, they simply add to the path. When we get back to the starting point the path is complete, but it still needs to be rendered. The stroke command causes the path to be drawn with the current line width, dash pattern, color or gray, miter limit, join, and cap. The stroke operator consumes the path as part of its operation, and initializes the path like the newpath operator does. At this point the stroked path is sitting in the image buffer, but not yet

printed. The final command in our example, showpage, forces the page to be printed.

Congratulations, you've completed your first PostScript program. Now let's go back and improve it. The first thing to do is add commands so that we can set the thickness of the path that strokes the rectangle. The stroke is always centered on the path and is rendered according to the current settings, even if the settings were different when the path was created. The operator that sets line thickness is called setlinewidth. It expects an number that is in decimal points (1/72th of an inch). To set a quarter-inch thick line for the box, we need to add 18 setlinewidth to our code at some point prior to the stroke command. Since the setlinewidth operator does not affect the building of the path, it is customary to place it right before the stroke command. So now our code looks like this:

```
newpath
100 100 moveto
200 100 lineto
200 200 lineto
100 200 lineto
100 100 lineto
18 setlinewidth
stroke
showpage
```

Now when we send the code down to the printer we get a box that looks like this. Notice the cutout in the lower left corner? This results from the way PostScript draws thick lines. In the default mode it starts at the start of the path and stops at the end of the path. Stopping at the end of the path leaves the cutout. We need the stroke to go a little further. The distance is equal to half the thickness of the stroke. But compensating for it in the program would be a nightmare as the distance is different if the lines meet at other than 90 degrees to each other.

But as mentioned before, PostScript is a language designed for graphics. Obviously this is a serious problem, so there is a specific answer, the closepath operator. The closepath operator is used in place of the last lineto. This tells PostScript to connect the current point to the first point and treat the join as if it were just another join in the path. This way the box ends up looking like we expect it no matter how thick the line is.

Next we want to fill the box with gray. Let's take the code we have so far and replace the stroke command with a fill operator. Additionally we will use the setgray operator to set

the level of gray we want. The `setgray` operator defines white as 1 and black as 0. So a dark gray is any number less than .5 and a light gray is above .5. In effect we are setting the percentage of white in the "ink" we will be using. Adding the `setgray` operator we have:

```
newpath
100 100 moveto
200 100 lineto
200 200 lineto
100 200 lineto
closepath
18 setlinewidth
.75 setgray
fill
showpage
```

The `setlinewidth` command does nothing in this case, but doesn't hurt anything so we leave it in. The result of the code is a light gray box.

Now let's tackle the big task, rendering a box with both a stroke and a fill. The simple answer would seem to be to add the `stroke` command back after the `fill` command. But if we do that the printer will only image the filled box. This is so because the path that was filled was consumed by the `fill` operator before the `stroke` operator could use it. Since there was nothing to stroke, nothing happened. There are two solutions, both of which work. The first is to describe the path again before using the `stroke` operator; this has the unfortunate side effect of doubling the size of the code. The second answer is to take advantage of the graphics stack. To do this we need two new operators, `gsave` and `grestore`. The `gsave` operator tells PostScript to save a copy of everything about the current graphics state on the graphics stack, except the graphics rendered so far. The current path is one of the things saved, so we can use it twice with the following code:

```
gsave
18 setlinewidth
stroke
grestore
.75 setgray
fill
```

When the `grestore` operator is executed, the original version of the current path is restored, even though the `stroke` command consumed it. An important facet of PostScript programming is to make your code as short as possible, and the `gsave/grestore` combination is a powerful tool in achieving that goal. With the new `gsave/grestore` code, our project looks like this:

```
newpath
100 100 moveto
200 100 lineto
200 200 lineto
100 200 lineto
```

```
closepath
gsave
18 setlinewidth
stroke
grestore
.75 setgray
fill
showpage
```

But if we swap the order of the `fill` and `stroke`, along with their support commands the output is changed.

This points out a very important feature/limitation of PostScript. Once a path has been rendered to the image buffer, it is there to stay. It can not be changed or manipulated in anyway, but it can be overwritten. In this respect PostScript is much like a paint program.

## In Comparison to C

Now that we are familiar with the basic operators and have a bit of a feel for how paths and graphics work, let's compare PostScript to 'C'. First off 'C' is a compiled language whereas PostScript is interpreted. The major repercussions of this is that PostScript is slower, but does not need to be compiled. The second difference is that PostScript uses a stack-based system to pass data to all its operators, even procedures. The third difference is that there is no concept of the entire source or object file being located somewhere. While there is looping, there are no labels or goto operators of any sort. A PostScript file is processed as it is received.

Other than these major differences, PostScript has much in common with other high level languages. As mentioned above, PostScript is designed to describe graphics, and that it does do well.

## Bigger and Better Things

Back to our sample code. Since we are mimicking the classic "Hello World" program, our code should say "Hello World" also. To do this we need to use some font-related operators. Due to the way PostScript handles fonts, this is likely to be much different than you are use to.

To image text, we must first find, scale, and set a font. The first operator, `findfont`, accepts the name of a font and tries to find it. If it can, it will place the instructions for the font on the dictionary stack. If it can not find the font, it will place the default font on the stack (usually Courier) and report an error. The instructions that are placed on the font stack are for a one-point (1/72th of an inch) version of the font. Since this is usually too small, we need to enlarge the font with the `scafont` operator. The `scafont` operator accepts a point size from the operand (main) stack and a font from the dictionary stack, if either is missing an error will occur. The results of the `scafont` operator is a scaled version of the font replaces the original one-point version on the dictionary stack. Now we are ready to use the font. Finally we use the `setfont` operator to pull the scaled font off of the fonts stack and use it as the current font. With the current font set, we can use the `show`

operator to render some text. The show operator accepts a PostScript string (surrounded by ( ) ) and does nothing but render the text in the image buffer according to the current settings, much like fill. But wait, where do we want the text to go? Using a moveto we can place the text inside the box. The following is the current code:

```
newpath
100 100 moveto
200 100 lineto
200 200 lineto
100 200 lineto
closepath
gsave
18 setlinewidth
stroke
grestore
.75 setgray
fill
/Helvetica findfont
18 scalefont
setfont
104 143 moveto
0 setgray
(Hello World) show
showpage
```

Notice that we have to use the setgray operator to set the current gray to black. If we don't, the text will render the same gray as the box fill and not be visible.

### Take Control

Text and graphics are the heart of what most people need PostScript for, but it's by no means all there is to PostScript. There are also operators that offer a certain amount of control over the execution of the operators. The first is the if operator. Much as you would expect it allows for the conditional execution of code. But we also said that there is no concept of blocks of code floating around, so how does it work? Using curly braces {} we can encapsulate a series of operands and operators and pass them as a block to the if operator. If we wanted to write only a line of text when the top of the operand stack is negative, we would write:

```
dup 0 lt {(Negative) show} if
```

Of course we may want to do something else if the number is positive, so we would use the ifelse operator and write the following:

```
dup 0 lt? {(Negative) show} {(Positive) show} ifelse
```

What is happening is that the interpreter treats the code in {} as a single object on the operand stack. Essentially it is a black box. When the if or ifelse operator decides which one to execute, it takes the code out of the braces and passes it to the interpreter.

There are also operators for looping—for, repeat, and loop. The repeat and loop operators are very similar in that they take a procedure, like the if operator, and have it executed repeatedly. In the case of repeat, a count is passed along with the operator. The loop operator executes until an exit operator is used. By comparison, for works more like a traditional for loop. A starting value, incrementing value and limit value are passed with the procedure. The control value is incremented, placed on the stack, and the code is execute until the limit value is reached. The placing of the current index on the stack is quite useful in many cases. For example, to add all the numbers from one to ten using each operator would take the following code to leave the answer on the stack:

```
0 1 1 10 {add} for
```

```
0 0 10 {exch 1 add dup 3 1 roll add} repeat exch pop
```

```
0 0 {exch 1 add dup dup 10 gt {exit} if 3 1 roll add} loop pop
```

This would seem to make a strong case for the for operator, but there are times when each operator has its strengths.

Looking back over the article, you can see that PostScript is very different from the programming languages you may be accustomed to. While we have not covered all of the drawing operators, you can also see why PostScript is said to describe a page rather than just drawing one. In the next part of this series we will look at how you can use PostScript with your program by constructing a PostScript header. In the final part of the series we will examine in detail the most widely-used PostScript header, the one for *Adobe Illustrator 88*, and how you can make use of it.

If you would like to know more about PostScript, I would highly recommend that you pick up the *PostScript Language Reference Manual, Second Edition*, and the *PostScript Language Tutorial and Cookbook*. The first book is the definitive specification of the PostScript language and the second is an excellent guide to learning PostScript.



Please write to:  
**Dan Weiss**  
**c/o AC's TECH**  
**P.O. Box 2140**  
**Fall River, MA 02722**

# —Nee Correction continued from page 4

```

moveq.b #1,aa          ;aa always at least 1
move.b 1(a4),d1         ;"top" neighbor value
beq.s #112,%           ;branch if 0
add.b d1,d0             ;add 1 to current value
cmpi.b #50,d1           ;is it 50?
beq.s #112,%           ;increase aa by 1
addq.b #1,aa
;2%
move.b 1(a4),d1
beq.s #113,%
add.b d1,d0
cmpi.b #50,d1
beq.s #113,%
addq.b #1,aa
;13%
move.b 1(a4),d1
beq.s #114,%
add.b d1,d0
cmpi.b #50,d1
beq.s #114,%
addq.b #1,aa
;14%
move.b 4(a4),d1
beq.s #115,%
add.b d1,d0
cmpi.b #50,d1
beq.s #115,%
addq.b #1,aa
;15%
divu aa,d0              ;total values/aa
cmp.b d1,d0             ;greater than 50-qg?
bgt.s #116,%
add.b gg,d0             ;ok to add qg
bra.s testdone%
;16%
moveq.b #50,d0          ;can't exceed 50
bra.s testdone%
dead%
moveq #0,d0             ;dead cell becomes healthy
testdone%
move.b d0,(a5)+         ;save new value;increase array
endm

pset macro
move.l rp,a1
move.w across,d0
add.w #xoff,d1          ;center across
move.w down,d1
add.w #yoff,d1          ;center down
ext.l d0
ext.l d1
gfxlib writepixel
endm

color macro
move.l rp,a1
gfxlib setapen
endm

array macro              ;(address,8(20-0))
move.l #size,d0
move.l #510004,d1
syslib allocmem
move.l d0,v1
beq v2
endm

```

```

even% mact%
ds.w
endm

start:
move.l rp,stack
movem.l a0/d0,-(rp)     ;save 1st parameter
address,length
open_libs:
openlib intuition, intbase,done
openlib graphics,gixbase,close_int

set_up:
make_screen:
openscreen myscreen,screen,close_libs
openwindow mywindow>window,close_screen
move.l d0,a0
movea.l ww,rport(a0),a1
move.l a1,rp
movea.l window,a0
intlib viewport,address
move.l d0,vp            ;viewport address

movea.l d0,a0
lea color_table,a1
moveq #16,d0            ;16 new colors
gfxlib loadrgb1

movea.l rp,a1
move.l #jam?,d0
gfxlib setdymd

get_bit_planes:
movea.l rp,a1
movea.l 4(a1),a1
lea bpl,a0
move.l 8(a1),a0)        ;bitplane address
move.l 12(a1),(a0)+
move.l 16(a1),(a0)+
move.l 20(a1),(a0)+
move.l 24(a1),(a0)

memory:
array array1,close_window
movea.l d0,a4
array array2,close_out

random:
syslib forbid
lea $df1000,a5
movea.l rp,a1
movea.l gfixbase,a0
moveq #0,down
r1 move.w 6(a5),d1       ;CIA register
moveq #0,across
r2 move.w 6(a5),d3
add.w d3,d4
add.b across,d4
cmpi.b #50,d4           ;maximum cell value
bhi.s r2
move.b d4,d0
move.b d4,sum
cmpi.b #0,d0            ;0 gets its own color
beq.s r2color
.rsr.b #2,d0            ;color/4
addq.b #1,d0

r2color:
color
pset
move.b sum,(a4)+
swp d4

```

```

addq.w #1,a4
cmp.w #len,a4
bne.s #0
addq.w #1,d0
cmp.w #len,d0
bne.s #0

get_cli_input:
    move.l 1sp,d1      ;parameter passed
    movea.l a0,a5
    move.l d0,d7      ;# of characters
    subq   #1,d7
    beq.s  default    ;no parameters
    move.b (a5),d8     ;get first value
    sub.l  #510,d8     ;and normalize it
    lea    1(a5),a1    ;skip space or comma
    subq   #7,d7
    move.b (a5)+,d1    ;get second value
    sub.l  #510,d1     ;and normalize it
    lea    1(a5),a5    ;skip space or comma
    subq   #7,d7
    move.b (a5)+,d2    ;get next value
    sub.l  #510,d2     ;and normalize
    subq   #1,d7
    beq.s  clidone     ;branch if no
    move.w #10,d3      ;that was 10's digit
    and.l  #001,d2     ;clear rest of word
    mulq.w d3,d2
    add.l  (a5),d2     ;add 1's digit
    sub.l  #510,d2     ;and normalize
    bra.s  clidone

default:
    move.b #2,d0       ;no default values
    move.b #3,d1       ;F1
    move.b #5,d2       ;G0

clidone:
    move.b d0,k1
    move.b d1,k2
    move.b d2,qg
    moveq.b #0,d0
    sub.l  qg,d0
    move.b d0,d11     ;no qg
    lea.l  ep1,a3

showit:
uleft:
    movea.l array1,a4
    movea.l array2,a5
    moveq   #0,d0
    move.b (a4),d0     ;first column
    test   len*lenml, lenml, len
    ;

trow:
    movea.l array1,a4
    lea    1(a4),a4     ;next row
    movea.l array2,a5
    lea    1(a5),a5
    moveq   #0,across
    move.w #lenm3,across ;number of times

trl:
    moveq   #0,d0
    move.b (a4),d0
    test   len*lenml, 1,1, len
    lea    1(a4),a4
    db:    across,trl

uright:
    movea.l array1,a4
    lea    lenml(a4),a4
    movea.l array2,a5

```

```

    lea    len1(a4),a1
    moveq   #0,d0
    move.b (a4),d0
    test   len*lenml, lenml, len

leftside:
    movea.l array1,a4
    lea    len(a4),a4
    movea.l array2,a5
    lea    len(a5),a5
    moveq   #0,d0
    move.w #lenml,d0
    ;
    moveq   #0,d0
    move.b (a4),d0
    test   mlen, lenml, 1, len
    lea    lenml(a4),a4 ;move to
    lea    lenml(a5),a5 ;right side
    moveq   #0,d0
    move.b (a4),d0
    test   mlen, 1, lenml, len
    lea    1(a4),a4
    db:    down, a4

;yet:
;lower left
    movea.l array1,a4
    lea    len*lenml(a4),a4
    movea.l array2,a5
    lea    len*lenml(a5),a5
    moveq   #0,d0
    move.b (a4),d0
    test   mlen, lenml, 1, len*lenml

brow:
;bottom row
    movea.l array1,a4
    lea    len*lenml(a4),a4
    movea.l array2,a5
    lea    len*lenml(a5),a5
    moveq   #0,across
    move.w #lenm3,across

browl:
    moveq   #0,d0
    move.b (a4),d0
    test   mlen, 1, 1, len*lenml
    lea    1(a4),a4
    db:    across, browl

right:
;lower right
    movea.l array1,a4
    lea    lenml*lenpl(a4),a4
    movea.l array2,a5
    lea    lenml*lenpl(a5),a5
    moveq   #0,d0
    move.b (a4),d0
    test   mlen, 1, mlenml, len*lenml

;er:
;center square
    movea.l array1,a4
    lea    lenpl(a4),a4
    movea.l array2,a5
    lea    lenpl(a5),a5
    moveq   #1,d0
    moveq   #1,across
    move.w #lenm3,down

;
    move.w #lenm3,across

;1
    moveq   #0,d0
    move.b (a4),d0
    test   mlen, 1, 1, len

```

```

lea    1(a4),a4
dbf    across,c1
lea    2(a4),a4
lea    2(a5),a5
dbf    down,l2

switch    ;(for slower computers)
movea.l array1,a4
movea.l array2,a5
move.w #lenm1,down
s1 move.l #len/4-1,across
s2 move.l (a5)+,(a4)+
dhf     across,s2
dbf     down,s1

: movea.l array1,a4
movea.l array2,a5
movea.l bp1,a1
lea     byte(a1),a1
move.l #lenm1,down
qp5
moveq   #wpl,across    ;length/32-1
qp4
moveq   #51f,d2        ;(1) all 32 bits
qp3
move.b  (a5)+,sum
; move.b sum,(a4)+
cmpi.b  #0,sum
bcq.s   qp1
asr.b   #2,sum
addq.b  #1,sum
qp1
asr.b   #1,sum          ;1st color bit to x
roxl.l  #1,d3           ; and to d3
asr.b   #1,sum          ;2d color bit to x
roxl.l  #1,d4           ; and to d4
asr.b   #1,sum          ;3d color bit to x
roxl.l  #1,d0           ; and to d0
asr.b   #1,sum          ;4th color bit to x
roxl.l  #1,d1           ; and to d1
dbf     d2,qp3          ;do 32 times
move.l  d1,$5dc0(a1)    ;all 4th color bits
move.l  d0,$3e80(a1)    ;all 3d color bits
move.l  d4,$1f40(a1)    ;all 2d color bits
move.l  d3,(a1)+        ;all 1st color bits
dbf     across,qp4
lea     byteoff(a1),a1
dbf     down,qp5
loop
bist    #6,$bfe001      ;pressed LMB?
bne     showit          ;branch if not

syslib   permit
close_mem:
movea.l array2,a1
move.l  #size,d0
syslib   freemem
close_out:
movea.l array1,a1
move.l  #size,d0
syslib   freemem
close_window:
movea.l window,a0
intl1b   closewindow
close_screen:
movea.l screen,a0
intl1b   closescreen
close_libs:
movea.l gfxbase,a1

```

```

syslib   closelibrary
close_int:
movea.l intbase,a1
syslib   closelibrary
done:
move.l   stack,sp
rts

evenpc

stack    dc.l 0
gfxbase  dc.l 0
intbase  dc.l 0
screen   dc.l 0
window   dc.l 0
sp        dc.l 0
vp        dc.l 0
array1   dc.l 0
array2   dc.l 0
bp1      dc.l 0
bp2      dc.l 0
bp3      dc.l 0
bp4      dc.l 0
bp5      dc.l 0
k1       dc.b 0

evenpc
k2       dc.b 0
evenpc
gg       dc.b 0
evenpc
d1f      dc.b 0
evenpc

graphics dc.b 'graphics.library',0
evenpc
intuition dc.b 'intuition.library',0
evenpc

myscreen
dc.w 0,0,320,200,depth
dc.b 0,1
dc.w 0
dc.w customscreen
dc.l 0,0,0,0
evenpc

mywindow
dc.w 0,0,320,200
dc.b 0,1
dc.l mousebuttons
dc.l borderless!activate!rmb
dc.l 0,0
dc.l 0
dc.l 0,0
dc.w 0,0,0,0
dc.w customscreen
evenpc

color_table:
dc.w $000,$12f,$24e,$36d
dc.w $48c,$5ab,$6ca,$7e9
dc.w $8e8,$9c7,$aag,$bcb
dc.w $c64,$d43,$e27,$f01
end

```

For any questions or comments regarding this correction  
or the article, write to

Bill Nee

c/o AC's TECH

P.O. Box 2140, Fall River, MA 02722

# **Technical Writers Hardware Technicians Programmers Amiga Enthusiasts**

Do you work your Amiga to its limits? Do you do create your own programs and utilities? Are you a master of any of the programming languages available for the Amiga? Do you often find yourself reworking a piece of hardware or software to your own specifications?

If you answered yes to any of those questions, then you belong writing for AC's TECH!

AC's TECH for the Commodore Amiga is the only Amiga-based technical magazine available! We are constantly looking for new authors and fresh ideas to complement the magazine as it grows in a rapidly expanding technical market.

Share your ideas, your knowledge, and your creations with the rest of the Amiga technical community—become an AC's TECH author.

**For more information, call or write:**

**AC's TECH  
P.O. Box 2140  
Fall River, MA 02722-2140**

**1-800-345-3360**

## example.c

14 /\* Start of Code Section: \*/

\* Example Machine File: example.c  
\* Copyright 1993, by J. T. Stachurski

\*\*\*\*\*  
\*\* UDAHR: Example program title

\*\*  
\*\* WARNING: Since all names generated by Flex used re  
directed input.

\*\* This example program can only be run from the  
CLI properly!

\*\* Unless you want to type the input title by  
hand!!

\*\* Use the Make File for info on how to run Example, for info  
Example.

\*\*  
\*\* See the Article for the syntax that Example expects!  
see in the

\*\* input file.

\*\*\*\*\*  
#include <stdio.h>  
#include <string.h>  
#include <ctype.h>  
#include <unistd.h>

void Ignore();  
void RptErr();  
void ShowNum();  
void DrawFunc();  
void MoveFunc();  
void OpenGraph();  
void ChkColor();  
void Cleanup();

/\* Bottom of Example \*/

/\*

SCANNER yytext;

MACHINE main;

RETURNS 0;

TYPE int;

TRANSITIONS

1 /\* Start of Transition table \*/

From State	What appears	action on
@Hires	@Move_	@Draw_
@LoRes		
@Color		
@Number	@Format	@EndLine:

>Begin< Done,Cleanup, Begin,Ignore, Begin,Ignore;  
Begin,OpenGraph;  
Begin,OpenGraph, DefMove,Ignore, DefDraw,Ignore;  
DefColor,Ignore;  
Error,RptErr, Begin,Ignore, Begin,Ignore;  
>DefColor< Error,RptErr, DefColor,Ignore;

Begin,Ignore, Error,RptErr;  
Error,RptErr, Error,RptErr, Error,RptErr;  
Error,RptErr;  
DefColor,ShowNum,DefColor,Ignore;  
DefColor,Ignore;  
>DefMove< Error,RptErr, DefMove,Ignore, Begin,MoveFunc,  
Error,RptErr;  
Error,RptErr, Error,RptErr, Error,RptErr;  
Error,RptErr;  
DefMove,ShowNum, DefMove,Ignore, Begin,Ignore;  
>DefDraw< Error,RptErr, DefDraw,Ignore, Begin,DrawFunc,  
Error,RptErr;  
Error,RptErr, Error,RptErr, Error,RptErr;  
Error,RptErr;  
DefDraw,ShowNum, DefDraw,Ignore, Begin,Ignore;  
>Error< Done,Cleanup, Begin,Ignore, Begin,Ignore;  
Begin,OpenGraph;  
Begin,OpenGraph, DefMove,Ignore, DefDraw,Ignore;  
DefColor,Ignore;  
Error,RptErr, Begin,Ignore, Begin,Ignore;  
>Done< Done,Cleanup, Done,Cleanup, Done,Cleanup;  
Error,RptErr;  
Error,RptErr, Error,RptErr, Error,RptErr;  
Error,RptErr;  
Done,Cleanup, Done,Cleanup, Done,Cleanup;

};

14 /\* Start of Code Section: \*/

extern char \*idstr;

int linecount = 0;

void Ignore() {  
if (isdigit(idstr[0]) && !isprint)  
linecount++;  
return;  
}

void RptErr() {  
fprintf(stderr, "ERROR in input stream\n");  
fprintf(stderr, "Found on line %d\n", linecount);  
return;  
}

struct NewScreen ns = {

0, 0, 640, 200, 4, 0xFF, 0xFF, HIRRES, CUSTOMSCREEN,  
NULL, (UBYTE \*) "StateMachine Example", NULL, NULL  
};

struct NewWindow nw = {

0, 0, 640, 200, 0xFF, 0xFF, CLOSEWINDOW,  
WINDOWCLOSE | ACTIVATE | SMART\_REFRESH,  
NULL, NULL, (UBYTE \*) "StateMachine Example",  
NULL, NULL, 20, 20, 640, 200, CUSTOMSCREEN  
};

extern struct InitializationBase \*InitBase;  
struct GfxBase \*GfxBase = NULL;

struct Window \*example\_window = NULL;  
struct Screen \*example\_screen = NULL;

void OpenGraph() {  
if (strcmp(idstr, "LORES") != 0)  
{  
ns.ViewModes = 0;  
ns.Depth = 5;



```

ns.Width      320;
ns.Height     200;
nw.Width      320; /* Make window 10x8 also! */
nw.Height     200;
nw.MaxWidth   320;
nw.MaxHeight  200;
}

if (!IntuitionBase) {struct IntuitionBase *}
    OpenLibrary("intuition.library", 0L
// -- NULL)
    return;
} { (GfxBase = (struct GfxBase *)
    OpenLibrary("graphics.library", 0L ))
NULL)
{
    CloseLibrary( IntuitionBase );
    return;
}
{ (example_screen = (struct Screen *) OpenScreen( &ns
// -- NULL)
{
    CloseLibrary( GfxBase );
    CloseLibrary( IntuitionBase );
    return;
}
nw.Screen = example_screen;
if (!example_window = (struct Window *) OpenWindow( &nw
// -- NULL)
{
    CloseScreen( example_screen );
    CloseLibrary( GfxBase );
    CloseLibrary( IntuitionBase );
    return;
}
return;

int  nums[5] = { -1, -1, -1, -1, 0 };

void  ShowNum()
{
    extern int atoi( char * );

    int index = 0;

    while (nums[index] >= 0)
        index++;
    nums[index] = atoi( idstr );
    return;
}

void  DrawFunc()
{
    if (example_window != NULL)
        Draw( example_window->RPort, nums[0], nums[1] );
    nums[0] = nums[1] = -1;
    return;
}

void  MoveFunc()
{
    if (example_window != NULL)
        Move( example_window->RPort, nums[0], nums[1] );
    nums[0] = nums[1] = -1;
    return;
}

void  ChgColor()
{
    extern struct ViewPort *ViewportAddress( struct Window * );

    if (example_window != NULL)
    {
        SetRGB4( ViewPortAddress( example_window ), nums[0],
nums[1],

```

```

        nums[2], nums[3] );
        *tApEn = example_window, &id, nums[4] );
        return;
    }

*tApEn: TRUE
#define TRUE 1
#define FALSE 0
#endif

void Cleanup()
{
    int      Checking = TRUE;
    ULONG    class = 0;
    struct IntuiMessage *message = NULL;
    char      n[128], *title = &n[0];

    if (example_window == NULL)
        goto SkipLoop;
    strcpy( title, nw.Title );
    strcat( title, "Click on Close window gadget to " );

    SetWindowTitles( example_window, title, 0 );

    while (Checking == TRUE)
    {
        if (message = (struct IntuiMessage *)
            GetMsg( example_window->UserPort ))
        {
            (void) Wait( 1L << example_window->UserPort
                >mp_SigBit );
            continue;
        }
        class = message->Class;
        (void) ReplyMsg( message );
        switch( class )
        {
            case CLOSEWINDOW:
                Checking = FALSE;
                break;
            default:
                break;
        }
    }

SkipLoop:

    if (example_window != NULL)
        CloseWindow( example_window );
    if (example_screen != NULL)
        CloseScreen( example_screen );
    if (GfxBase != NULL)
        CloseLibrary( GfxBase );
    if (IntuitionBase != NULL)
        CloseLibrary( IntuitionBase );
    return;
}

/* End of Code Section */

```



Please write to  
**J.T. Steichen**  
 c/o AC's TECH  
 P.O. Box 2140  
 Fall River, MA 02722

# Maintain your edge ...

*AC's TECH* provides you with advanced insight into Amiga technology; now subscribe to the magazines that will always keep you up-to-date on the newest Amiga products and late-breaking Amiga news.

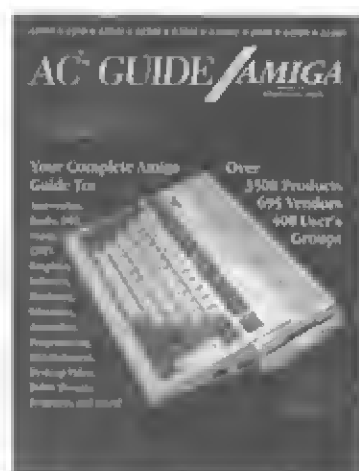
Subscribe to the best resources available for the *AMIGA*

# Amazing Computing



*Amazing Computing*, the first monthly Amiga magazine, remains the first in new product announcements, unbiased reviews, and consistently in-depth reporting. AC's unique columns like *Roomers* and *BugBytes*, step-by-step programming articles, and entertaining tutorials have made it the magazine of choice with devoted Amiga fans. With AC you remain on the cutting edge of Amiga product development.

## AC's GUIDE



*AC's GUIDE* remains the world's best resource for Amiga product information. A compilation of new product announcements from AC and exhaustive research, *AC's GUIDE* is a constantly updated reference to the ever-changing Amiga market.

With an AC SuperSub, you will receive 12 issues of *Amazing Computing* and two issues of *AC's GUIDE* at a tremendous savings.

## AC's TECH



AC's *TECH* was the first disk-based technical magazine for the Amiga. This quarterly collection of programs, techniques, and developer issues has been created for the Amiga owners who want to do more with their Amigas. If you want to expand your Amiga knowledge beyond the ordinary, then AC's *TECH* is a must.

Complete your *Amazing Computing* library\* and FRS collection.

\*while supplies last

Mail or fax (508-675-6002) the enclosed order form or call toll-free in U.S. or Canada, 800-345-3360. Foreign orders please call 508-678-4200.

Mail to: PIM Publications Inc., P.O. Box 2140, Fall River, MA 02722

**YES!** The "Amazing" AC publications give me **3 GREAT** reasons to save!

Please begin the subscription(s) indicated below immediately!

Name \_\_\_\_\_

Address \_\_\_\_\_

City \_\_\_\_\_ State \_\_\_\_\_ ZIP \_\_\_\_\_

Discover Visa MC # \_\_\_\_\_

Expiration Date \_\_\_\_\_ Signature \_\_\_\_\_

Please circle to indicate this is a **New Subscription** or a **Renewal**



Call now and use your  
Visa, Master Card, or  
Discover or fill out and  
send in this order form!

1 year of AC	12 big issues of Amazing Computing! Save over 43% off the cover price!	US \$27.00 <input type="checkbox"/> Canada/Mexico \$34.00 <input type="checkbox"/> Foreign Surface \$44.00 <input type="checkbox"/>
1-year SuperSub	AC + AC's GUIDE—14 issues total! Save more than 45% off the cover prices!	US \$37.00 <input type="checkbox"/> Canada/Mexico \$54.00 <input type="checkbox"/> Foreign Surface \$64.00 <input type="checkbox"/>
1 year of AC's TECH	4 big issues of the FIRST Amiga technical magazine with Disk!	US \$43.95 <input type="checkbox"/> Canada/Mexico \$47.95 <input type="checkbox"/> Foreign Surface \$51.95 <input type="checkbox"/>

Please call for all other Canada/Mexico/foreign surface & Air Mail rates.

Check or money order payments must be in US funds drawn on a US bank; subject to applicable sales tax.



NAME \_\_\_\_\_

ADDRESS \_\_\_\_\_

CITY \_\_\_\_\_ STATE \_\_\_\_\_ ZIP \_\_\_\_\_

DISCOVER \_\_\_\_\_ VISA \_\_\_\_\_ M/C # \_\_\_\_\_

EXPIRATION DATE \_\_\_\_\_ SIGNATURE \_\_\_\_\_

**Amazing Computing Back Issues** \$5.00 each US, \$6.00 each Canada and Mexico,

\$7.00 each Foreign Surface. Please list issue(s) \_\_\_\_\_

**Amazing Computing Back Issue Volumes:**

Volume 1—\$19.95\* Volume 2, 3, 4, 5, 6, or 7—\$29.95\* each

\*All volume orders must include postage and handling charges: \$4.00 each set US, \$7.50 each set Canada and Mexico, and \$10.00 each set for foreign surface orders. Air mail rates available.

**AC TECH/AMIGA**

Single Issues Just \$14.95!

Pick any 4 issues for only \$45.00!

V1.1 (Pre-release), V1.2, V1.3, V1.4,

V2.1, V2.2, V2.3, V2.4, V3.1, V3.2



**Freely Distributable Software — Subscriber Special (yes, even the new ones!)**

1 to 9 disks	\$6.00 each
10 to 49 disks	\$5.00 each
50 to 99 disks	\$4.00 each
100 or more disks	\$3.00 each

\$7.00 each for non subscribers

(three disk minimum on all foreign orders)

**Amazing on Disk:**

AC#1 Source & Listings V3.8 & V3.9	AC#2 Source & Listings V4.3 & V4.4
AC#3 Source & Listings V4.5 & V4.6	AC#4 Source & Listings V4.7 & V4.8
AC#5 Source & Listings V4.9	AC#6 Source & Listings V4.10 & V4.11
AC#7 Source & Listings V4.12 & V5.1	AC#8 Source & Listings V5.2 & V5.3
AC#9 Source & Listings V5.4 & V5.5	AC#10 Source & Listings V5.6 & V5.7
AC#11 Source & Listings V5.8, 5.9 & 5.10	AC#12 Source & Listings V5.11, 5.12 & 5.1
AC#13 Source & Listings V6.2 & 6.3	AC#14 Source & Listings V6.4, 6.5
AC#15 Source & Listings V6.6, 6.7, 6.8, 6.9	

Please list your Freely Redistributable Software selections below

**AC Disks** \_\_\_\_\_

(numbers 1 through 15)

**AMIGAs** \_\_\_\_\_

(numbers 1 through 26)

**Fred Fish Disks** \_\_\_\_\_

(numbers 1 through 860)

**Complete Today, or telephone  
1-800-345-3360 now!**

You may FAX your order to 1-508-675-6002

Please allow 4 to 6 weeks for delivery of  
subscriptions in US.

(Domestic and Foreign air mail rates available on request)

Check or money order payments must be in US funds drawn on a US bank; subject to applicable sales tax.

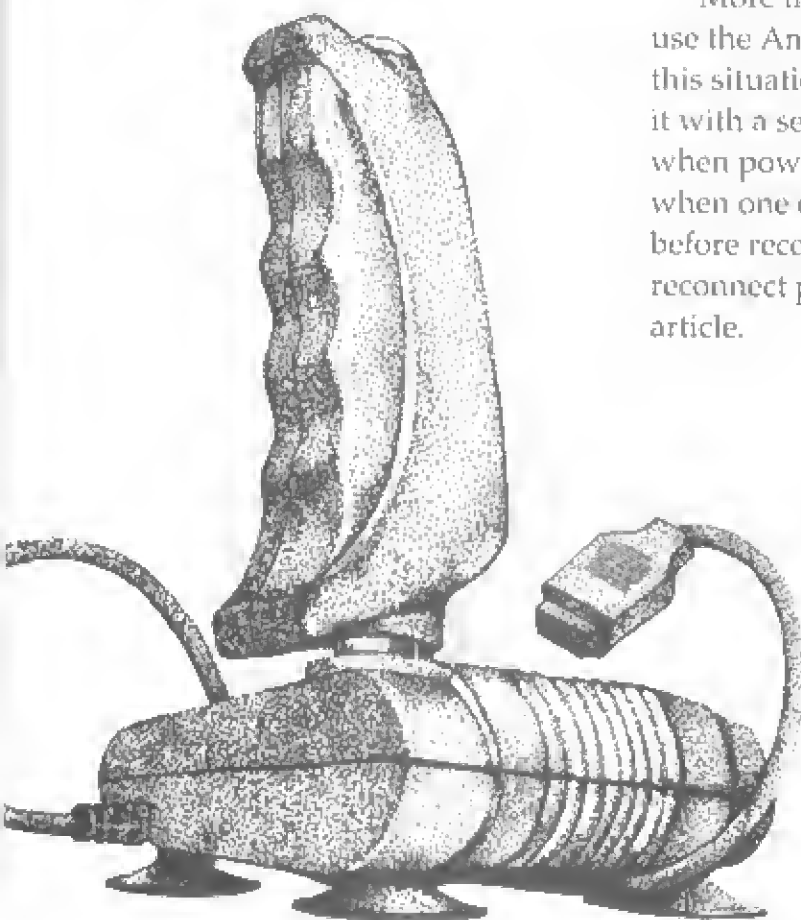


**SUBSCRIBE!**

**ORDER!**

# Need an Extra Joystick Port?

More than anything else, my two young children like to use the Amiga to wage competition against each other. For this situation, they must disconnect the mouse and replace it with a second joystick, not a recommended operation when power is on. Near disaster struck, almost inevitably, when one of my children forgot to switch the computer off before reconnecting the mouse. If you have this disconnect-reconnect problem, you may be interested in the following article.



Any electronic project or project which may or may not require modification of your Amiga should only be attempted by experienced persons. AC's TECH is not responsible for any damage to your Amiga or any other equipment that may occur as a result of this project.

There are only two ports on your Amiga, one for the mouse and the other for a joystick. But sometimes you may want to play a game against another human, not only against the computer. To switch devices safely, you must switch the Amiga off or you may have the problem I had. I had to replace a microfuse which protects the mouse port on my Amiga 2000. This is not an easy job because the fuse is soldered to the motherboard. One must open the case, remove the motherboard, and resolder a new microfuse.

To protect my computer, I considered buying a switch costing around \$40, but I enjoy small electronic projects, so I decided to design my own with two extra gadgets that the other switches don't have.

My switch is different from the commercial ones I have seen. Mine has no button for choosing between the mouse and the second joystick. When you want to use the second joystick, you just push the fire button on it; when you want the mouse back, you simply push the left button on it. The switching is done by the electronics inside the box without mechanical parts.

Another difference with my switch: it has a rapid-fire function for the first joystick, the one you use when playing single. So with my switch, I have two functions combined in the same box for less than the price of a store-bought.

# Build One!

*by Jaques Halleé*

The parts I used are easy to find at any electronic components supply. And if you follow the instructions, you will be able to build yours during the next weekend.

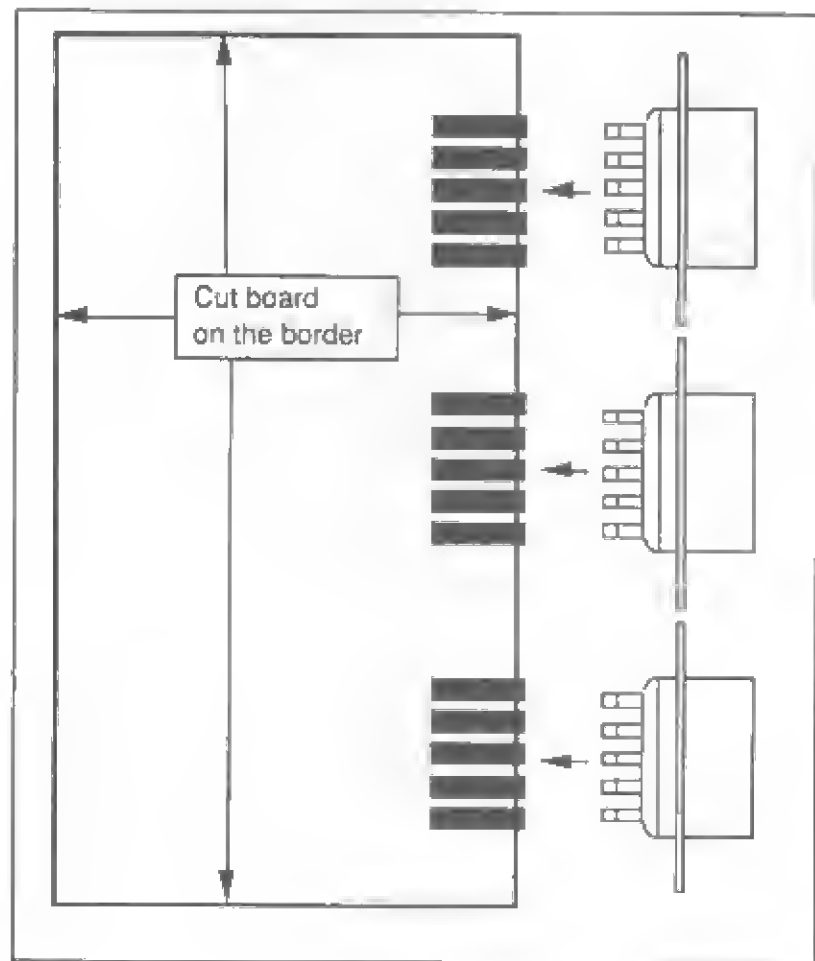
## How It Works

Please refer to the schematic in Figure 1 for the description of the parts and their functions. There are two sections in the unit: the first section is the switch itself and the second section is the rapid-fire function.

The first section can be divided in three blocks. The first block is represented by U3, the selector. The selector is equivalent to four switches. One side of the switch is connected to the mouse and the other side is connected to joystick 2. The output is connected to the Amiga port #1, normally the mouse port. The selector is activated by the second block which is represented by U2, the latch. The latch itself is activated by the left mouse button or the fire button of the joystick. The latch acts as a memory; as long as you don't touch the fire button of the joystick, the output of the latch remains low and the selector outputs the movement of the mouse. If you touch the fire button on the joystick, the output of the latch flips to a high level and the selector outputs the movement of the joystick. The third block of this section consists of one gate U4 to combine (logic OR)

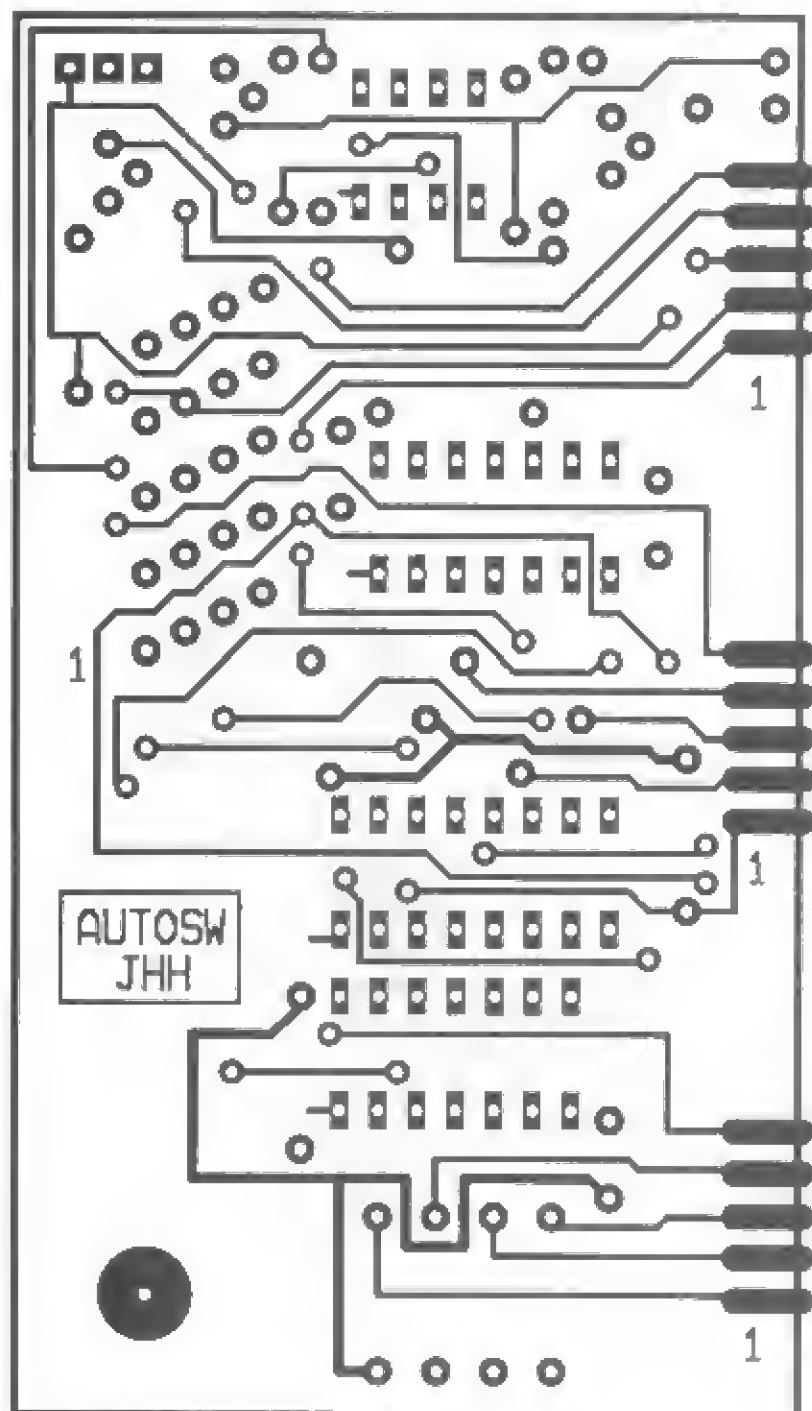
the action of the left mouse button and the fire button of the joystick.

The second section of the unit is the rapid-fire function for joystick 1. The heart of this part is the timer U1. It is an oscillator producing a train of pulses as long



Right: Figure 5. Not to scale.

Figure 5



as fire is pressed. You can adjust the pulse rate, or frequency, by turning POT1. One pulse is equivalent to one hit on the fire button. The transistor Q1 senses the fire button and starts the oscillator if held down. Transistor Q2 takes the output of the oscillator and feeds it to the input of the Amiga pin. Switch SW1 is there to select the normal function of the fire button or the rapid-fire mode.

### First, Buy All the Components.

For a complete list of parts, see the Parts Table. The three ICs (U2,U3,U4) are "HCTLS" and could be replaced by "HC" family instead of "HCTLS" but not by regular "LS" or "S" family because of their power consumption.

I chose a tantalum capacitor type for C2 because of the small size; you may choose any type of capacitor with the same value, but be careful of the size. All the components are standard parts and you should not have any difficulties finding them. When they are available at a Radio Shack, I give the catalogue number because these stores are well known. If you shop carefully, the parts should not cost more than \$20 plus the cost of the PCB.

Also, if you are not interested in the rapid-fire function, omit the parts referring to this section. The switch should work on any Amiga including the new A4000. I have already tested it on A500, A1000, A2000 and A3000.

### Then Install the Parts

First, identify which side is the components side, the side where you insert the parts, and which one is the solder side. Refer to Figure 2 for the components side (top) and Figure 3 for solder side (bottom). It is very important that you insert the parts on the good side (top).

For those of you who are making the PCB, there are some points to consider. *Figures 2 and 3 are double size*

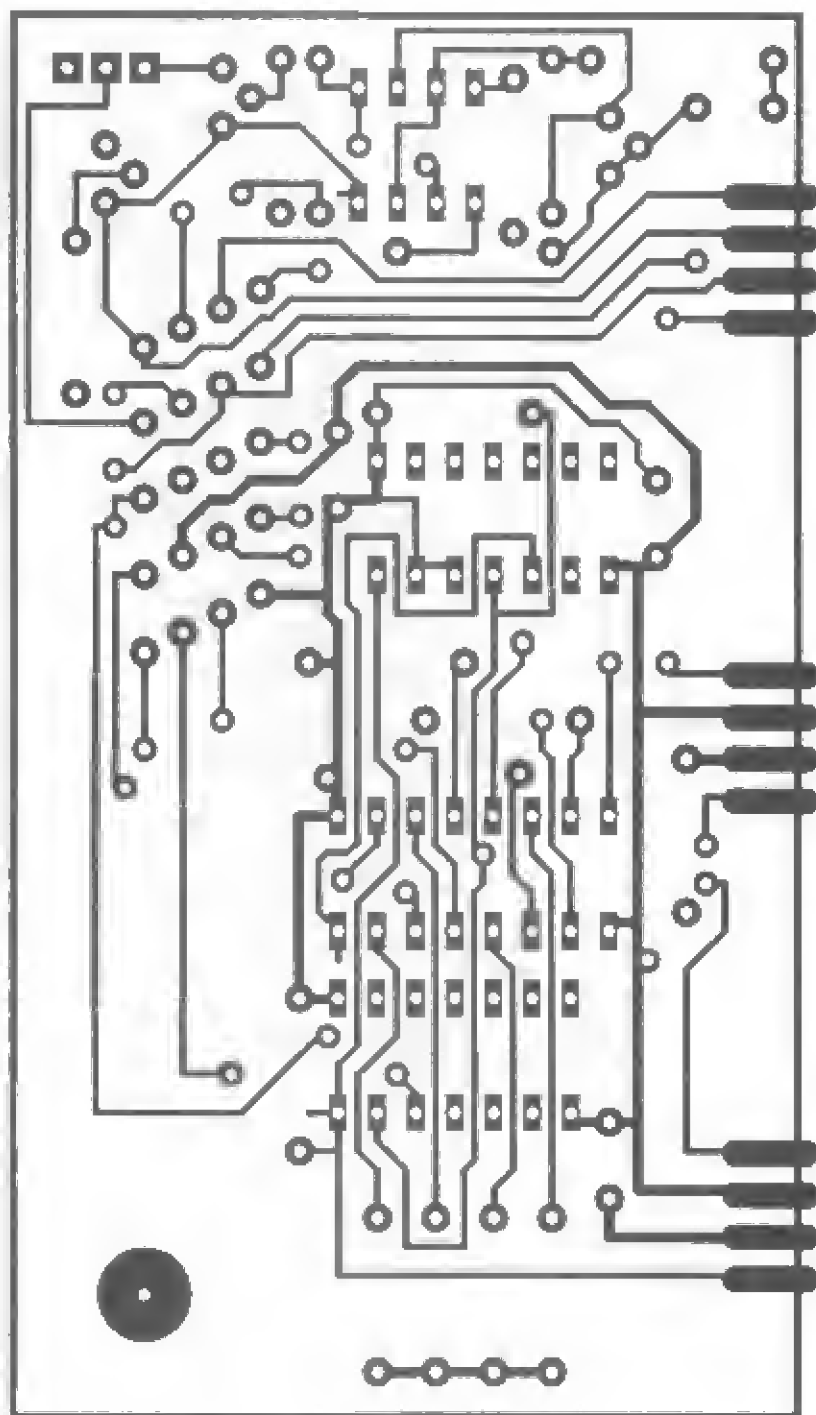
**Left: Figure 2. Image is 2x actual size. If you use this figure as a template, remember to reduce it by 50%!**

(2X). Make a negative of these two figures and reduce to half size. Then align the two negatives carefully with the letter on top; you should see "JHH." When your PCB is done you have to cut it on the border (cut off the border) as shown in Figure 5. The last thing to do before soldering parts is to drill the holes at the right size. If you drill all the holes at 0.028 inch, you should be able to insert all the parts.

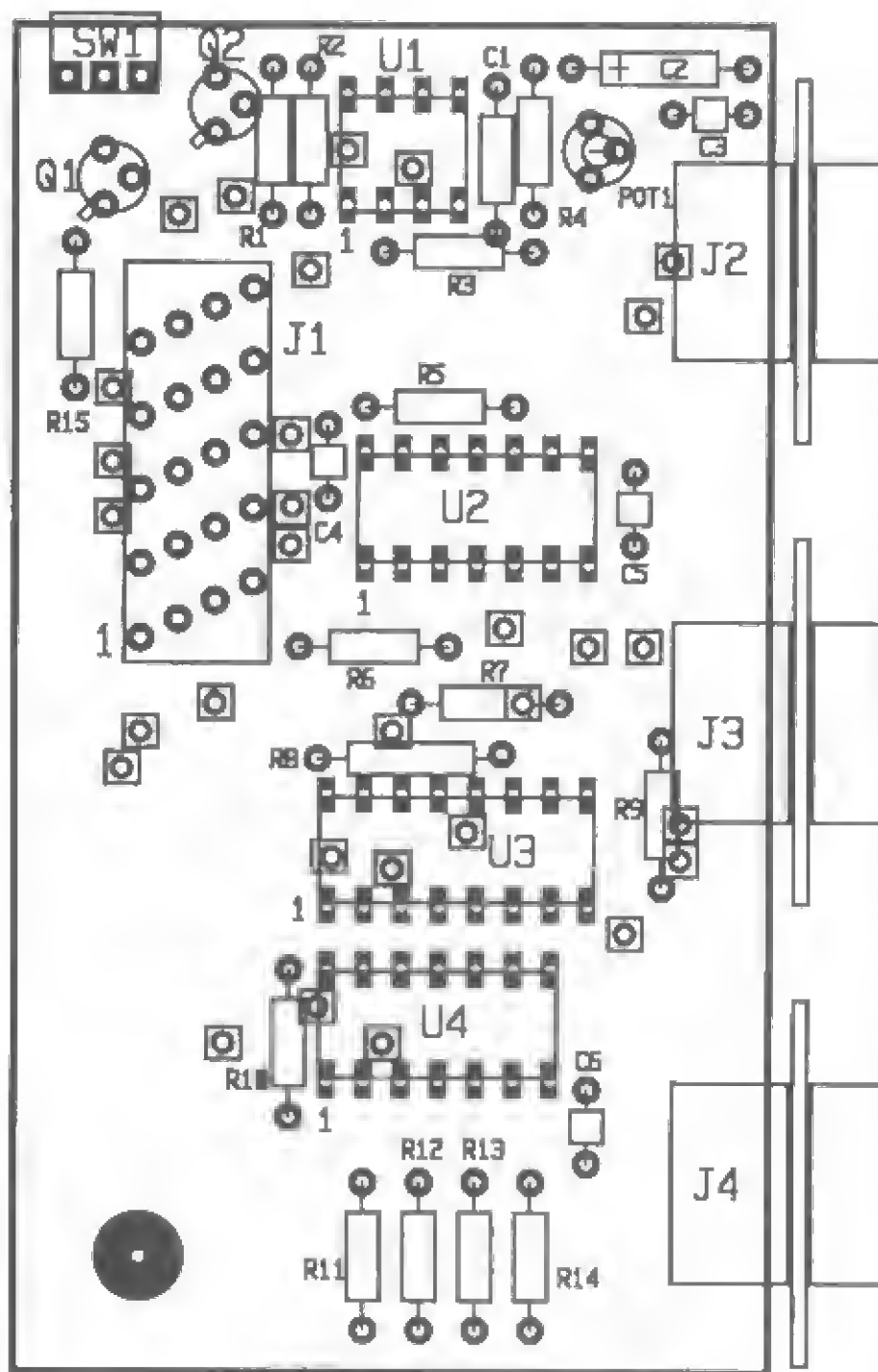
Now you have your PCB ready. The first step before installing the components is to solder the feed-thru. I call a feed-thru a piece of wire soldered on both sides to make a connection from one side of the PCB to the other; refer to Figure 6. There are 30 feed-thrus to make before installing components. The feed-thrus are easy to locate: the pad for a feed-thru is smaller than the others. Refer to Figure 4 for the component locations you can see feed-thrus with a square over the pad. Before soldering your first feed-thru, install the PCB on two spacers at both ends. The thickness of the spacers can be about 1/8 inch.

The easiest way of soldering a feed-thru is to take a piece of solid wire, #22. Then you skin two or three inches off it and you insert it into a hole, solder it on one side, cut it, and do the same with all the feed-thrus for that side. After, you flip the board over and solder all the feed-thru on the other side. To assure a good solder, you must put the iron tip on one side of the wire and the solder on the opposite side. When you see the solder melting all around the pad, you remove the iron quickly to prevent overheating. Make sure that you cut the wire as close as possible to the PCB; refer to Figure 6.

When you have built all the feed-thrus, you are ready to install the rest of the components. The following list shows the order that I recommend for



Right: Figure 3. Image is 2x actual size. If you use this figure as a template, remember to reduce it by 50%!



soldering the components:

- 1- sockets
- 2- resistors
- 3- capacitors
- 4- transistors
- 5- potentiometer
- 6- DB-9 male
- 7- wires and switch SW1
- 8- transition connector and the flat cable

### Sockets

When you install the sockets, you must take care of the direction. Refer to Figure 4 for the good socket direction. All sockets have a way to identify pin 1; find it and install the socket with pin 1 facing the transition connector J1, as shown on Figure 4. If you want your socket to remain in place while soldering, just bend one pin at each corner after insertion into the PCB, then solder sockets on the bottom side only. Don't insert the ICs into the sockets right now.

### Resistors

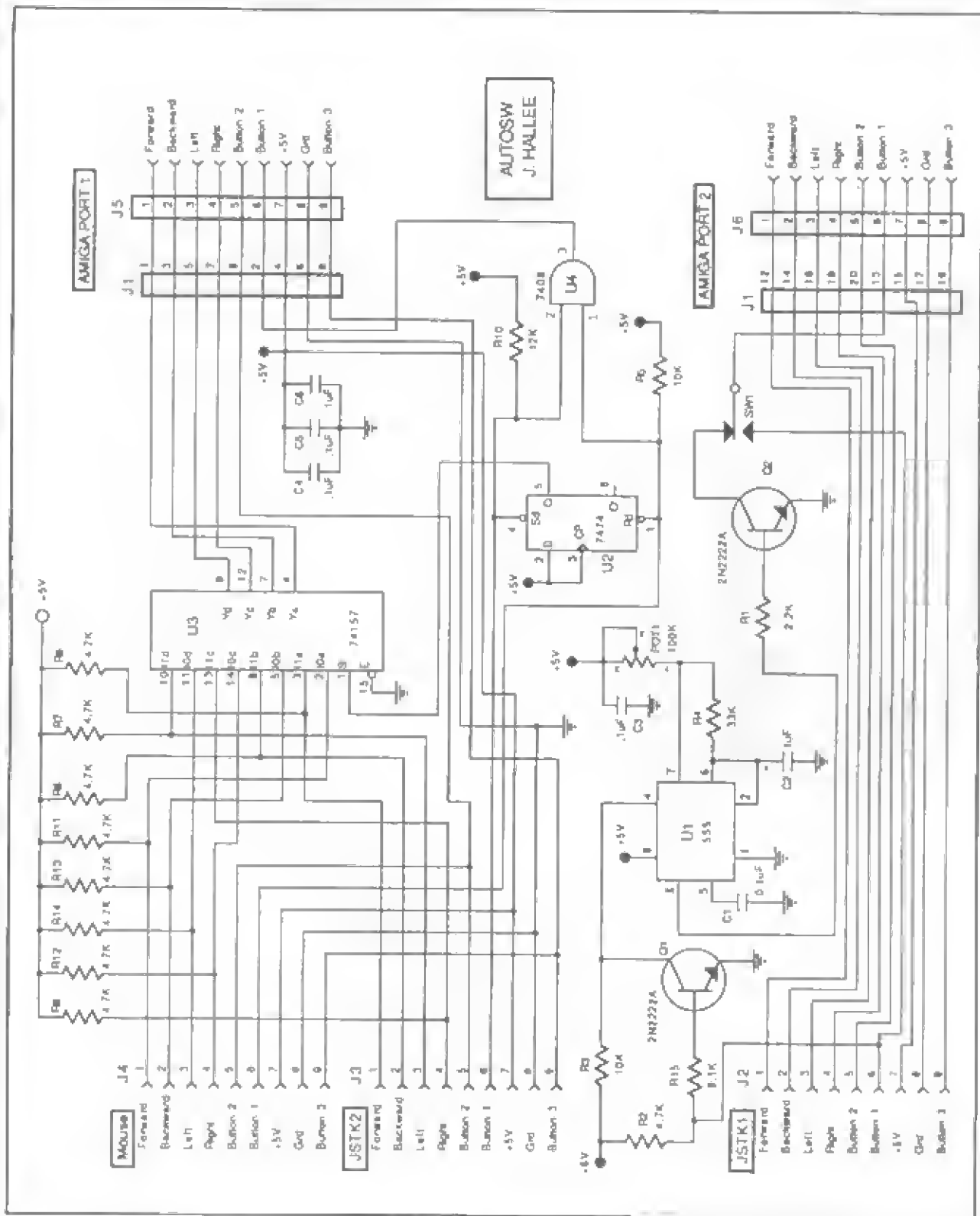
Check the color of the resistors before installing them on the PCB. The way you read the value of a resistor is simple: there are four color bands on the resistor. One band should be gold (5% tolerance), turn the resistor to put the gold band on the right side. Refer to the parts list for the color corresponding to each value. For all the resistors, you must solder them on both sides of the PCB, because some connections are also made on top.

### Capacitors

Except for C2, there is no preferred way for mounting the

Left: Figure 4. Image is 2x actual size. If you use this figure as a template, remember to reduce it by 50%! Right: Figure 1. This schematic is not to scale.





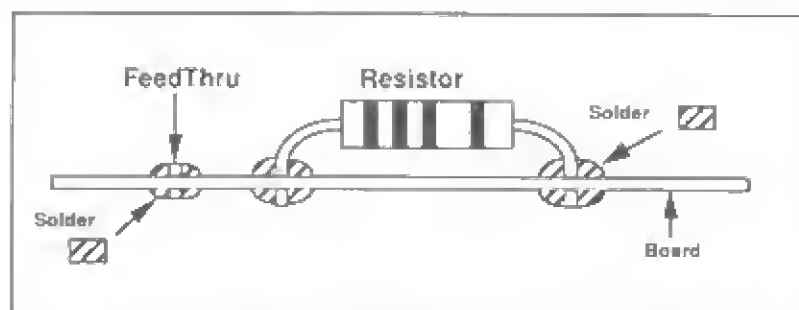


Figure 6

Figures 6 & 7

An extra  
joystick port  
can reduce  
wear and  
tear on your  
existing ports.

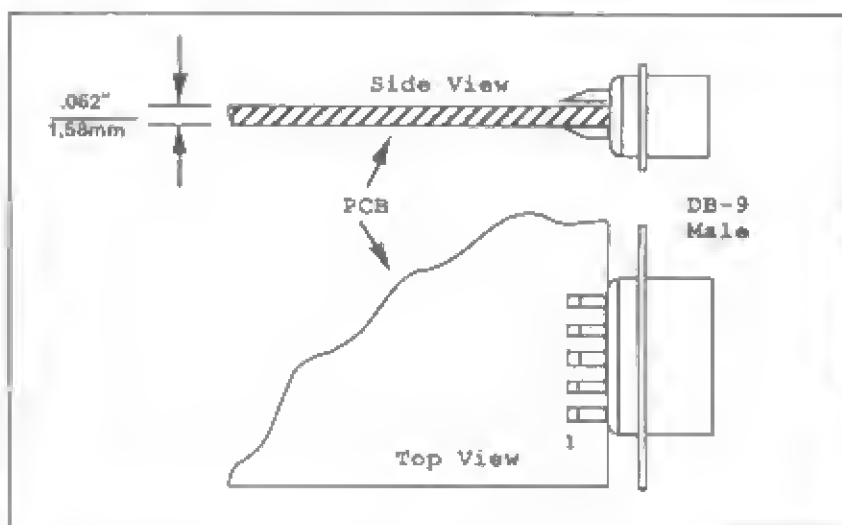


Figure 7

capacitors. C2 should have a "+" sign to identify the polarity. Make sure that you install it with the sign in the good direction. As for the resistors, you must solder the capacitors on both sides of the PCB.

### Transistors

There are two types of transistor you can buy. The first type has a metal case and it is round with a little tab to identify the emitter. Make sure that the tab is on the good direction as shown in Figure 4. The other type has a plastic case and there is a flat on it. When you look at the transistor with the pins down and the flat in front, you should read the pins from left to right. The emitter is left, base is center and collector is right. Make sure that the emitter is on the tab side in Figure 4. For the plastic transistor, you will have to bend the center pin to insert it into the PCB. For both types of transistors, leave at least 3/8 inch between the transistor and the PCB. You

must solder them on both sides of the PCB to ensure the connection with the top conductors.

### Potentiometer

If you use the potentiometer from Radio Shack, you will have to bend the pins before inserting it into the PCB. Use a pair of long-nose pliers to bend the pins. Identify the center pin; it should go into the center hole. You can also use another type of potentiometer and drill it hole in the box so you will be able to adjust the frequency externally without opening the box. For this, you will have to solder three wires from the PCB pads (SW1) to the potentiometer pins on the box. Remember, the center wire (slider) must be soldered to the center pad on the PCB.

### DB-9 male

You have to slide the PCB between the two rows of

Figures 8 & 9.

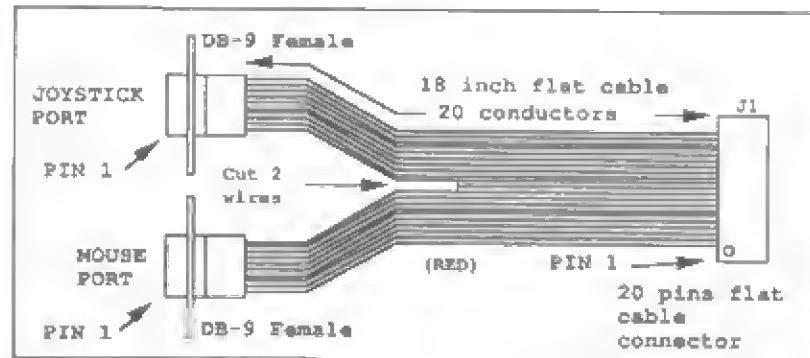


Figure 8

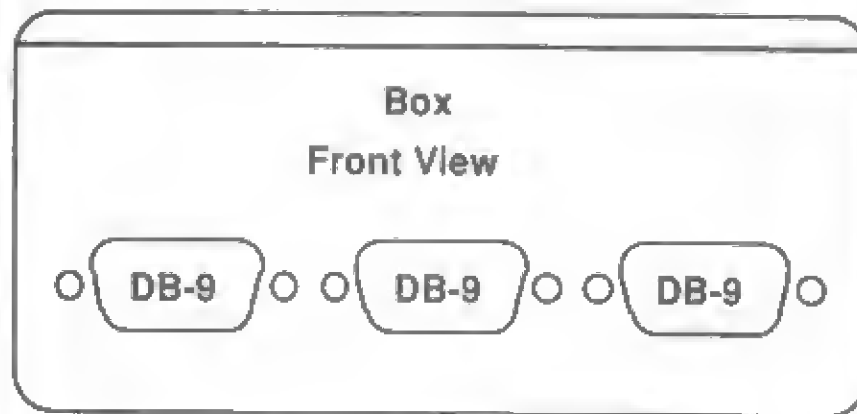


Figure 9

**Commercial  
switches  
of this  
type  
run \$40  
or more.**

pins of the connector. The top row has five pins. The bottom row has only four pins, and there is only one way to install it. Refer to Figure 7 for the installation of these connectors. Install the three connectors before soldering. The alignment of these connectors is critical; center the pins on the pads of the PCB. They should be pressed completely against the PCB. A little trick after you have inserted the connectors is to put the PCB vertically on a table using the connectors as a support. It is much easier this way to align the connectors. The space between the three connectors should be the same. When you are sure the connectors are correctly seated, then solder them to the finger pads.

### Switch SW1

Cut three #22 wires six inches long and then solder them on the PCB pads marked "SW1." Don't use solid wire; it is too easy to break when you bend it several

times. Then solder the other end of the wires on the switch. Make sure the center wire is soldered on the center pin of the switch. Make sure than you solder them on both sides of the PCB

### Transition Connector J1

For the transition connector you have two parts, one with the pins and the other is the cover (J1). Solder the pins part on the bottom side of the PCB only. Then install one end of the 20 conductors flat cable between the two parts of the connector. The red wire must be on the right side as shown in Figure 8. Use a small vise to crimp the flat cable into the connector. Be careful with the alignment of the flat cable and the connector.

Then at the other end of the flat cable, install the two DB-9 female connectors. Next, remove an eight-inch section of the two center wires. There should remain nine conductors on each side of the flat cable.

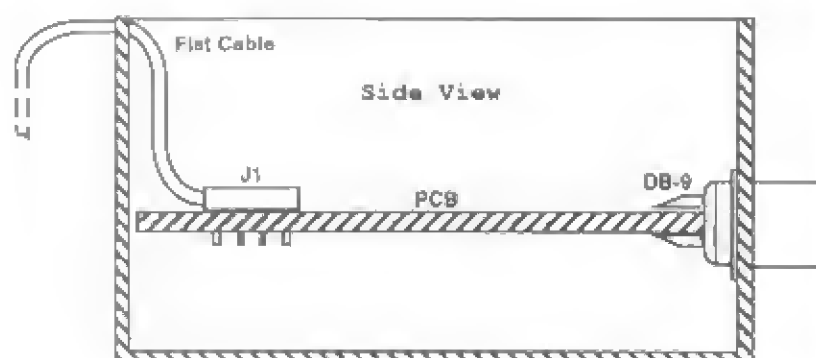


Figure 10

Left: Figures 10 & 11.  
Opposite: Table of Parts.

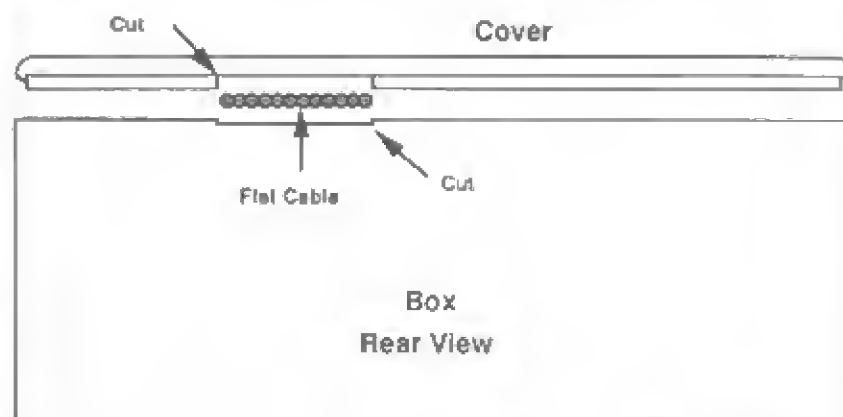


Figure 11

Crimp the two connectors DB-9 on each part of the flat cable. Take care with pin 1 of the connector; it should match with the red wire for one side. Then the other connector should go with the pin 1 on the center of the flat cable; refer to Figure 8. Now you are ready to test the unit.

Before connecting the unit to your Amiga, inspect the PCB and your solder joints carefully. If you have a magnifying glass, use it to check if you forgot a solder spot, or if you have shorts between two pads or two traces. This operation is very important and could prevent damage to your computer. The time you spend on inspection could save you lots of trouble and time debugging your unit. If you have done a good job soldering the parts, the unit should work right away. The most common problem is a forgotten solder joint on the top side of the board.

After the inspection, you are ready to connect it to your computer without the chips in the sockets. Be sure

your computer is *off*! Remove the mouse and the joystick if you have one already connected. Then connect the unit to your computer; the connector with the red wire should go to the mouse port. Turn your computer on and with a multimeter check if you have +5 volts on each socket according to the schematic. If you have all the sockets with the +5V and your computer still boots, then an important part is done. Otherwise, turn your computer off immediately and double-check your PCB, especially the trace with the +5V on it.

With your computer still off, install all the chips in their appropriate sockets and connect the mouse to the unit in the left connector; refer to Figure 13. You can also connect your joysticks to the other connectors. Turn your computer on again. Your pointer should move with the mouse as before.

If the mouse is OK, then you can first test-fire the button of the joystick, which is connected in the center connector. This action should freeze the mouse. Now

### Parts list for AUTOSW

Qty	Reference	Description
1	U3	74HCTLS157
1	U2	74HCTLS74
1	U4	74HCTLS08
1	U1	555 Timer
2	Q1,Q2	Transistor 2N2222A
3	J2,J3,J4	DB-9 Male, RS #276-1427
2	J5,J6	DB-9 Female flat cable
1	J1	20 pins flat cable transition connector (see note below)
1	POT1	Micro-potentiometer 100 Kohms, RS #271-284
1	SW1	Switch SPDT RS #275-662
1	R2	4.7 Kohms 1/4W 5% (yellow, purple, red, gold)
4	R6-R9	4.7 Kohms 1/4W 5% (yellow, purple, red, gold)
4	R11-R14	4.7 Kohms 1/4W 5% (yellow, purple, red, gold)
2	R3,R5	10 Kohms 1/4W 5% (brown, black, orange, gold)
1	R10	12 Kohms 1/4W 5% (brown, red, orange, gold)
1	R4	33 Kohms 1/4W 5% (orange, orange, orange, gold)
1	R1	2.2 Kohms 1/4W 5% (red, red, red, gold)
1	R15	9.1 Kohms 1/4W 5% (white, brown, red, gold)
1	C2	1.0 uF Tantalum
1	C1	0.1 uF Mylar
4	C3-C6	0.1 uF Decoupling capacitor
2		sockets 14 pins
1		socket 16 pins
1		socket 8 pins
1		small plastic box, RS #270-222
1		18 inches flat cable 20 wires

**NOTE:** For J1, you can buy it from the next three companies or an equivalent.

- Robinson Nugent #IDT-C20-1-T
- Ansley #609-2003
- 3M #3422-0000T

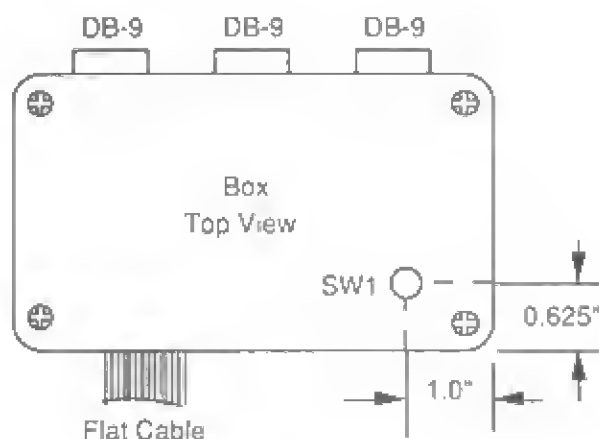


Figure 12

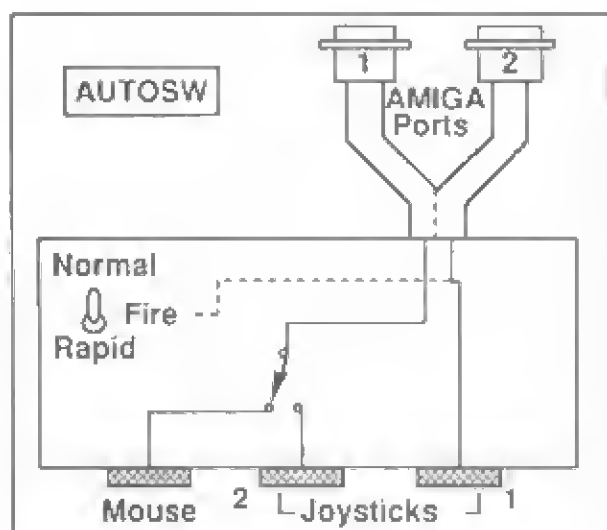


Figure 13

press on the left mouse button and your mouse should work again.

It is now time to check the two joysticks. Boot your computer with a game using two joysticks. Press the fire button on the central joystick and try to move your object in each direction with one joystick; then try the second joystick.

The last thing you have to check is the rapid-fire function of the joystick on the right connector. Boot a game with shooting action like *SideWinder*. Adjust the potentiometer in the center position with a small screwdriver and then press the fire button and hold it. The rapid-fire function should be active; if this is not the case, change the state of switch "SW1" and try again.

When you have found which position is rapid and which is normal mode, adjust the potentiometer for speed you prefer.

If everything works OK the first time, that means you have done a good job. Then turn your computer off. Disconnect the unit, the mouse, and the joysticks. Now you are ready to put it in a box.

### Put the Unit in a Box

The box I used is the Radio Shack #270-222 plastic box. You can use anything you like, but the rest of this section concerns this type of box. I designed the PCB to fit tightly in the box, there is not much lost space. I always like to keep things as small as possible.

The first thing to do on the box is to remove the slides inside the box on one side. You will drill the holes for the DB-9 into this side. I used a small cutter to remove these slides. Then you can finish the job with a piece of sand paper.

Next cut the three holes for the DB-9 male connectors. The easiest way is to draw the pattern for these connectors on a piece of thick paper. Put the PCB with the connectors face down on the paper. While holding the board with one hand, trace a line around each connector with a pencil. Then cut the inside part with a sharp knife. With this template, it will be easier to mark the holes on your box. The three holes must be centered on the box; check Figure 9 for more details. After you have cut three holes for DB-9 connectors, insert the three connectors externally into the box; with this way, you will be able to mark the six small holes for the screws on DB-9. Then you can install the board inside the box.

If you put six screws to hold the DB-9, it will be enough to hold the PCB inside the box. The next thing to do is to cut the edge of the box and the cover for the flat cable; refer to Figure 10 and 11.

The next operation is to drill a hole for switch SW1 inside the cover. Refer to Figure 12 for marking this hole. The size of the hole depends on the size of the switch you chose. If you have the one specified in the parts list, the hole will be 1/4 inch. After you have drilled the hole, install the switch according to Figure 13. You can take a reduced photocopy of Figure 13 and glue it to the cover.

I hope you and your computer will feel better when young kids want to play a game. If you have any questions or comments, please write.

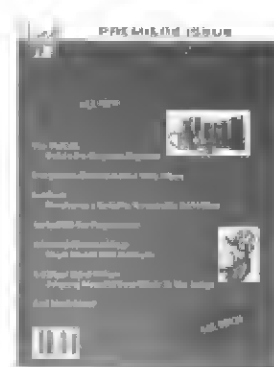


Please write to:  
Jacques Halleé  
c/o AC's TECH  
P.O. Box 2140  
Fall River, MA 02722

# AC's TECH Back Issue Index

## AC's TECH Volume 1 Number 1

Adapting Mattel's Power Glove to the Amiga by Paul King and Mike Cargal  
 AmigaDOS for Programmers by Bruno Costa  
 AmigaDOS, EDIT and Recursive Programming Techniques by Mark Pardue  
 An Introduction to Interprocess Communication with ARexx by Dan Sugalski  
 An Introduction to the libm.library by Jim Fiore  
 Building the VidCell 256 Grayscale Digitizer by Todd Elliott  
 Creating a Database in C, Using dBC III by Robert Broughton  
 FastBoot: A Super BootBlock by Dan Babcock  
 Magic Macros with ReSource by Jeff Lavin  
 Silent Binary Rhapsodies by Robert Tiess  
 Using Intuition's Proportional Gadgets from FORTRAN 77 by Joseph R Pasek



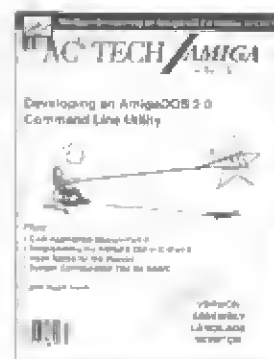
## AC's TECH Volume 1 Number 2

A Mega and a Half on a Budget by Bob Blick  
 Accessing Amiga Intuition Gadgets from a FORTRAN Program Part II-Using Boolean Gadgets by Joseph R. Pasek  
 Adding Help to Applications Easily by Philip S. Kasten  
 CAD Application Design: Part I - World and View Transforms by Forest W. Arnold  
 Interfacing Assembly Language Applications to ARexx by Jeff Glatt  
 Intuition and Graphics in ARexx Scripts by Jeff Glatt  
 Programming the Amiga's GUI in C Part I by Paul Castonguay  
 ToolBox Part I: An Introduction to 3-D Programming by Patrick Quaid  
 UNIX and the Amiga by Mike Hubbard



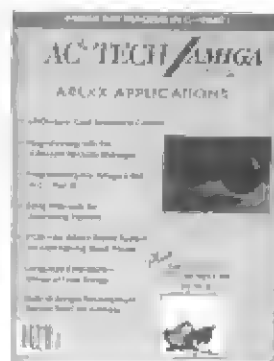
## AC's TECH Volume 1 Number 3

Accessing the Math Co-Processor from BASIC by R. P. Haviland  
 C Macros for ARexx by David Blackwell  
 CAD Application Design Part II by Forest W. Arnold  
 Configuration Tips for SAS-C by Paul Castonguay  
 Hash for the Masses: An Introduction to Hash Tables by Peter Dill  
 Programming for HAM-E by Ben Williams  
 Programming the Amiga's GUI in C-Part II by Paul Castonguay  
 The Development of an AmigaDOS 2.0 Command Line Utility by Bruno Costa  
 Using RawDofmt in Assembly by Jeff Lavin  
 VBRMon: Assembly Language Monitor by Dan Babcock  
 WildStar: Discovering An AmigaDOS 2.0 Hidden Feature by Bruno Costa



## AC's TECH Volume 1 Number 4

GPIO Low-Cost Sequence Control by Ken Hall  
 Language Extensions: Strings of Type Strings by Jimmy Hammonds  
 Programming the Amiga's GUI in C Part III by Paul Castonguay  
 Programming with the ARexxDB Records Manager by Benton Jackson  
 State of Amiga Development Denver DevCon Address by Jeff Scherb  
 STOX: An ARexx-Based System for Maintaining Stock Prices by Jack Fox  
 The Development of a Ray Tracer Part I by Bruno Costa  
 The Varafire Solution: Build Your Own Variable Rapid-Fire Joystick by Lee Brewer  
 Using Interrupts for Animating Pointers by Jeff Lavin



# COMPLETE VOLUMES!



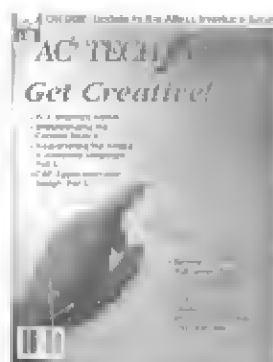
## AC's TECH Volume 2 Number 1

- AudiolProbe-Experiments in Synthesized Sound with Modula 2 by Jim Olinger
- CAD Application Design Part III by Forest W. Arnold
- Implementing an ARexx Interface in Your C Program by David Blackwell
- Low-Level Disk Access in Assembly by Dan Babcock
- Programming a Ray Tracer in C Part II by Bruno Costa
- Programming the Amiga in 680x0 Assembler Part I by William P. Nee
- Programming the Amiga's GUI in C Part IV by Paul Castonguay
- Spartan: Build Your Own SCSI Interface for Your Amiga 5000/1000 by Paul Harker
- The Amiga and the MIDI Hardware Specification by James Cook
- Writing Protocols for MusicX by Daniel Barrett



## AC's TECH Volume 2 Number 2

- Amiga Voice Recognition by Richard Horne
- Animated Busy Pointer by Jerry Trantow
- Blit Your Lines by Thomas Eshelman
- Copper Programming by Bob D'Asto
- Dynamically Allocated Arrays by Charles Rankin
- Implementing an ARexx Interface in Your C/Program Part II by David Blackwell
- Iterated Function Systems for Amiga Computer Graphics by Laura Morrison
- Keyboard I/O from Amiga Windows by John Baez
- MenuScript by David Ossorio
- Programming the Amiga in Assembly Language Part II by William P. Nee



## AC's TECH Volume 2 Number 3

- Backup\_DOC by Werther Pirani
- CAD Application Design Part IV by Forest W. Arnold
- HighSpeed Pascal by David Czaya
- PCX Graphics by Gary L. Falt
- Programming the Amiga in Assembly Language Part III by William P. Nee
- Programming the Amiga's GUI in C Part V by Paul Castonguay
- Understanding the Console Device by David Blackwell

## AC's TECH Volume 2 Number 4

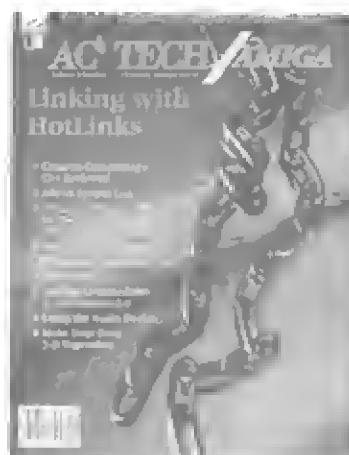
- Advanced Scripting by Douglas Thain
- Entropy in Coding Theory by Joseph Graf
- Fast Plots by Michael Griebeling
- Getconfirm() by John Baez
- In Search of the Lost Windows by Phil Burke
- No Mousing Around by Jeff Dickson
- Programming the Amiga in Assembly Language Part IV by William P. Nee
- Putting the Input Device to Work by David Blackwell
- Quarterback 5.0 a Review by Merrill Callaway
- Tape Drives by Paul Cittings
- The Joy of Sets by Jim Olinger
- TrueBASIC Extensions by Paul Castonguay





## ***AC's TECH* Volume 3, Number 1**

Comcast Computing's C++ by Forest Arnold  
ARexx System Log by Paul Gittings  
Bitmapped Graphics by Dan Weiss  
Transformer by Laura Morrisson  
Programming the Amiga in Assembly Language Part V by William P. Nee  
Trading Commodities in Workbench by Scott Palmateer  
Using the Audio Device by Douglas Thain  
Mike Your Own 3-D Vegetation by Laura Morrisson  
Linking with HotLinks by Dan Weiss

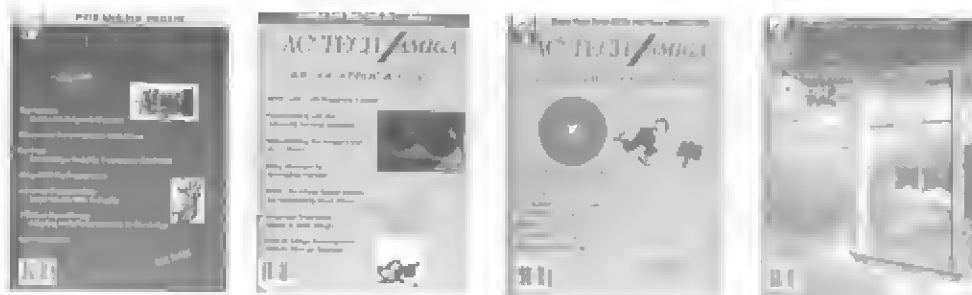


## ***AC's TECH* Volume 3, Number 2**

Ole by Thomas J. Eshelman  
Programming the Amiga in Assembly Language Part VI by William P. Nee  
Porting a B+Tree Library to the Amiga by John Bushakra  
Wrapped Up with True BASIC by Dr. Roy M. Nuzzo  
Assembly Language & Computer Simulations by William P. Nee  
Getfile Shell for True BASIC by Will Steinsek  
ARexx Disk Cataloger by T. Darrel Westbrook



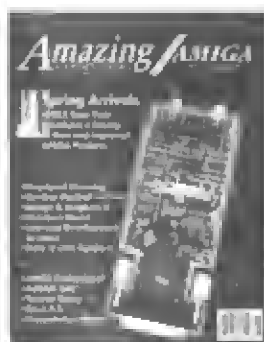
# **Thirsting for Amiga technical knowledge?**



**Let AC's TECH quench your thirst!**

**AC's TECH, an oasis of  
Amiga Technical Information**

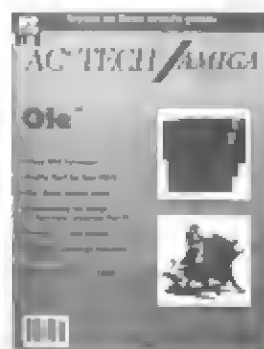
# *Amazing Computing Gives You Great Reasons to Own an Amiga*



Amazing Computing provides its readers with:

- In-depth reviews and tutorials
- Informative columns
- Latest announcements as soon as they are released
- Worldwide Amiga Trade Show coverage
- Programming Tips and tutorials
- Hardware Projects
- The latest non-commercial software

## *AC's TECH Gives You Great Reasons to Get Into Your Amiga*



AC's TECH offers these great benefits:

- The only disk-based Amiga technical magazine
- Hardware projects
- Software tutorials
- Interesting and insightful techniques and programs
- Complete listings on disk
- Amiga beginner and developer topics

---

*Order a SuperSub and get this great Amiga peripheral*



- World's best authority on Amiga products and services
- Amiga Dealers swear by this volume as their bible for Amiga information
- Complete listings of every software product, hardware product, service, vendor, and even user groups
- Directory of Freely Redistributable Software from The Fred Fish Collection

***12 Issues of Amazing + 2 AC's GUIDES!***

**SUMMER 1993 AC's GUIDE IS NOW AVAILABLE!**



# High Resolution Output

from your AMIGA™  
DTP & Graphic Documents

You've created the perfect piece, now you're looking for a good service bureau for output. You want quality, but it must be economical. Finally, and most important...you have to find a service bureau that recognizes your AMIGA file formats. Your search is over. Give us a call!

We'll imageset your AMIGA graphic files to RC Laser Paper or Film at 2400 dpi (up to 154 lpi) at an extremely competitive cost. Also available at competitive cost are quality Dupont ChromaCheck™ color proofs of your color separations/films. We provide a variety of pre-press services for the desktop publisher.

**Who are we?** We are a division of PiM Publications, the publisher of *Amazing Computing for the Commodore AMIGA*. We have a staff that *really* knows the AMIGA as well as the rigid mechanical requirements of printers/publishers. We're a perfect choice for AMIGA DTP imagesetting/pre-press services.

*We support nearly every AMIGA graphic & DTP format as well as most Macintosh™ graphic/DTP formats.*

*For specific format information, please call.*

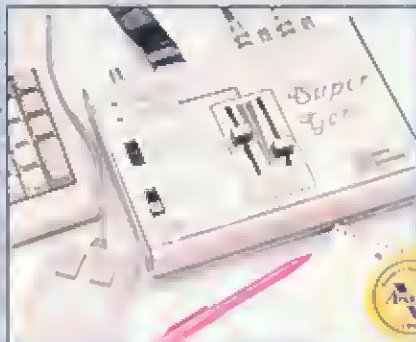
*For more information call 1-800-345-3360*

*Just ask for the service bureau representative.*

## SuperGen

### GENLOCK AND OVERLAY SYSTEM

- Only broadcast quality genlock for less than \$1000
- Two independent dissolve controls
- Software controllable
- Compatible with all Amiga models
- Notch filter



- The industry standard - yet to be equaled

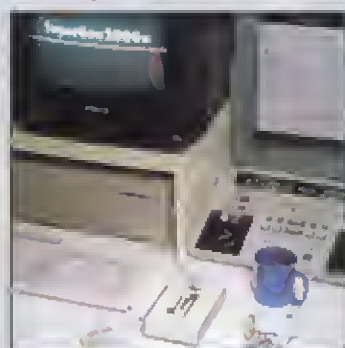
**NEW PRICE!**

**SuperGen**  
**\$549.00**

## SuperGen2000

THE FIRST TRUE Y/C GENLOCK AND OVERLAY CARD FOR THE AMIGA 2000 SERIES COMPUTER

- S-VHS, ELD-BETA, Hi8 compatible
- Broadcast quality NTSC RS-170A output
- SC/H phase adjustability
- Built-in sync generator
- Two independent dissolve controls



**NEW PRICE!**

**SuperGen 2000s**  
**\$1195.00**



THE FUTURE IS HERE!

Create spectacular true color animations on your Amiga.



Now Available in PAL

Paint, digitize and display beautiful full color composite video images on any Amiga

Capturing an image in 10 seconds from any color video camera or stable video source.

Full-featured paint, digitize and conversion software included

Compatible with AGA 1200 and 4000 Amigas in NTSC/PAL modes. Two to four times the speed of AGA animations (DCTV vs. HAM8) with greater color and resolution.

Compatible with all popular 3D, rendering, and graphics packages including:

AD-Pro, Aladdin 4D, AmigaVision, Brilliance, Calligari, Cinemorph, Draw4D, ImageMaster, Imagine, LightWave, MorphPlus, Real 3D, Scala, Scenery Animator, Sculpt, VistaPro, and many others...

**DCTV**  
(NTSC or PAL)  
**NEW PRICE!** **\$299.00**

## RGB CONVERTER



Allows the use of DCTV with standard RGB monitors (1084) in standard NTSC or PAL modes. Also permits the use of external genlocks like our SuperGen.

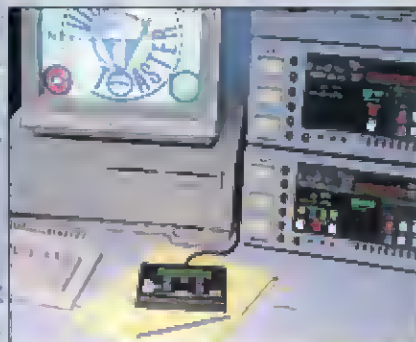
**NEW PRICE!**

**RGB Converter**  
**\$199.00**

## The Kitchen Sync

HOT NEW PRICES

### TWO CHANNEL TBC SYSTEM



The Kitchen Sync provides two channels of time base correction - the perfect low cost TBC solution for the Video Toaster™.

With a Video Toaster, the Kitchen Sync provides a complete A/B roll editing system.

Two complete infinite window time base correctors on one IBM AT/Amiga compatible card.

- Absolute 100% broadcast quality
- Compositing of Y/C video in
- Includes easy to use external control panel
- No waveform monitor needed
- Variable speed strobe
- Freeze Frame, two rock-solid Freeze Fields
- Low power consumption
- Lowest TBC price per channel
- Works with consumer grade VCRs

**Kitchen Sync**  
**NEW PRICE!** **\$1295.00**

## Genlock Option

Required to synchronize the Kitchen Sync to an external video source.

**Genlock Option**  
**NEW PRICE!** **\$150.00**

## S-VHS Option

Required to enable S-VHS/Hi-8 (Y/C) video outputs.

**S-VHS Option**  
**NEW PRICE!** **\$99.00**



**FREE SHIPPING**



on all VISA & MC orders in the US.

Next Day Shipping add \$5.00. COD - Cash only - add \$10.00.

Call by 2:00pm PST/5:00pm EST for same day shipping.

**DIGITAL**

Worldwide Distributors and Dealers Wanted. Inquiries invited.

**CALL**  
**DIGITAL**  
**DIRECT**  
**1-800-645-1164**  
**Orders only**

9:00am to 5:00pm PST M-F

For technical information call 916-344-4825

CREATIONS P.O. Box 97, Folsom CA 95763-0097 • Phone 916-344-4825 • FAX 916-635-0475

SuperGen, SuperGen2000s, DCTV, DCTV RGB Converter, and Kitchen Sync are trademarks of Digital Creations, Inc. Video Toaster is a trademark of Newtek, Inc. IBM and IBM AT are registered trademarks of IBM, Inc. Amiga is a registered trademark of Commodore-Amiga, Inc.